Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl für Regelungstechnik
Prof. Dr.-Ing. G. Roppenecker Prof. Dr.-Ing. Th. Moor

Projektarbeit

# Supervisor synthesis for discrete event systems with colored marking: Algorithmic Implementation and Case Studies

Als Projektarbeit
vorgelegt von

**Matthias Singer**

| Betreuer: | Betreuer: | Ausgabedatum: | 01.03.08 |
|---|---|---|---|
| Dr.-Ing. Klaus Schmidt | Prof. Dr.-Ing. Th. Moor | Abgabedatum: | 18.07.08 |

# Supervisor synthesis for discrete event systems with colored marking: Algorithmic Implementation and Case Studies

Aufgabenstellung:

Recently, efficient methods for the supervisory control of multitasking discrete event systems (MDES) have been developed. In contrast to the classical supervisory control theory by Ramadge/Wonham, MDES are modeled by colored marking generators, that is, automata where states can have multiple colors that represent distinct tasks. The main goal of the controller design is then to find supervisors that are nonblocking with respect to each individual color.
In this thesis, the advantages of multitasking supervisory control shall be investigated. In the first step, algorithmic support for the synthesis of multitasking supervisors has to be provided. To this end, the C++-software library libFAUDES which has been developed at the Chair of Automatic Control of the University of Erlangen-Nuremberg shall be used. The original library supports the analysis of discrete event systems (DES) and the synthesis of supervisors for DES according to the supervisory control theory by Ramadge/Wonham. In this work, a plugin for multitasking supervisory control has to be implemented. Second, a study of two example systems has to be carried out. The cat-and-mouse-in-a-maze example shall be extended to two connected floors, and modular multitasking supervisory control has to be applied to solve control problems that involve multiple tasks for the cat and the mouse. Finally, hierarchical multitasking supervisory control shall be applied to the model of a manufacturing system with several components, and the benefit of using multiple colors has to be discussed.

Es wird ausdrücklich auf die „Richtlinien zur Anfertigung von Studien- und Diplomarbeiten“ hingewiesen.

(Prof. Dr.-Ing. Th. Moor) (Dr.-Ing. K. Schmidt)

Matthias Singer

# Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 18. Juli 2008

Matthias Singer

# Contents

**1 Introduction** **1**

**2 Definitions** **5**

2.1 Formal Languages and Automata . . . 6

2.2 Natural Projection . . . 7

2.3 Accessibility and Coaccessibility . . . 9

2.4 Blocking . . . 10

2.5 Colored Parallel Composition . . . 11

2.6 Multitasking Supervisory Control . . . 12

2.7 Multitasking Hierarchical and Decentralized Control . . . 13

**3 Multitasking Plugin for the libFAUDES Software Library** **17**

3.1 Plugin Description . . . 18

3.2 Representation of Colors . . . 19

3.3 Class “mtcGenerator” . . . 21

3.3.1 Interface Methods . . . 21

3.3.2 Implementation Details . . . 22

3.4 Further Functions . . . 24

3.4.1 Deterministic . . . 24

3.4.2 Project . . . 25

3.4.3 Parallel . . . 26

3.4.4 SupConNB . . . 28

3.4.5 Statemin . . . 29
3.4.6 UniqueInit . . . 31

**4 Examples** **32**

4.1 Cat and Mouse in a Maze . . . 32
4.1.1 Description of the Example and the Requirements . . . 32
4.1.2 Modeling and Specification . . . 34
4.1.3 Computing the supervisor . . . 36
4.2 Fischertechnik Production Plant . . . 37
4.2.1 General Description of the Production Plant and the Chosen Part for this Example . . . 37
4.2.2 Modeling and Specification . . . 41
4.2.3 Hierarchical Structure . . . 45

**5 Conclusion** **50**

**Bibliography** **52**

**Appendix** **54**

**A Fischertechnik Production Plant** **54**

A.1 Models and Specifications for Conveyor Belt 4 and its additional components . . . 54
A.2 Hierarchy Diagrams . . . 59

# Chapter 1

# Introduction

Various technical systems such as communication networks, manufacturing systems or decentralized sensor-actuator-systems can be modeled as *discrete event systems (DES)*. For this reason, the modeling of DES and the investigation of their behavior has attracted a lot of interest during the last two decades. New methods to control and verify discrete event systems have been developed.

The initial effort has been made by P.J. Ramadge and W.M. Wonham [RW87], who published their work on a "Supervisory Control Theory" in the late 1980s. They introduced minimal restrictive supervisors which may constrain a plant's behavior by disabling controllable events in order to comply with a given specification. In this context, the specification both expresses the sequences of events that are allowed and marks certain tasks that have to be achieved in the controlled behavior.

The computation of minimally restrictive supervisors from a given plant model and a specification can be carried out by computational tools such as the software library libFAUDES which is developed by the Discrete Event Systems group of the Chair of Automatic Control at the University of Erlangen-Nürnberg. It is a universally applicable program library for DES written in C++ and its sources are freely available [lib08] under the terms of the GNU Lesser General Public License [lgp07].

Considering the standard supervisory control theory, there are two main difficulties which have to be overcome.

First, the number of system states increases enormously for systems that consist of a growing number of components. This phenomenon is also known as "state space explosion" [RW87]. It results in a very high complexity of the regarded models, such that computing those models leads to an enormous memory consumption. Consequently, only small

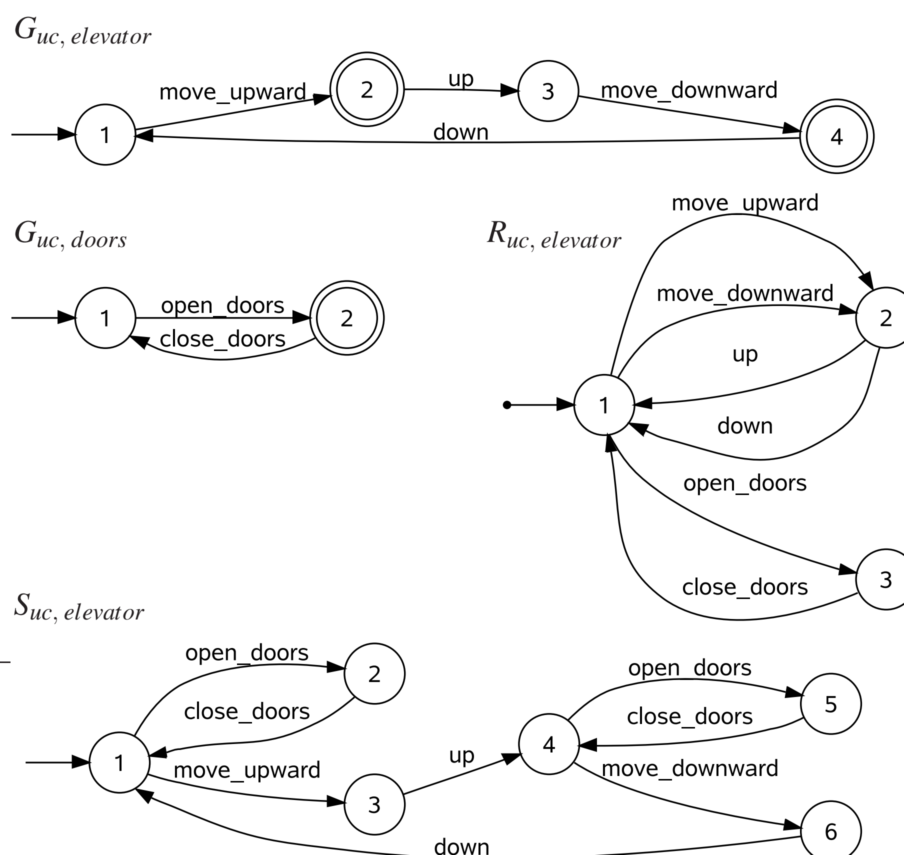


Figure 1.1: Model, specification and supervisor for a simple elevator

models can be examined, which is in contradiction to the desire for a practical application.

This problem can be diminished by using hierarchical abstraction. Models and specifications are divided into several little parts, which are combined step by step to an entire model. In doing so, several levels of abstraction are obtained. Events that are not important to the next higher level can there be omitted. As a result, the highest level automaton's size is significantly smaller than the one of a monolithic automaton [Sch05].

Second, there often are systems that require the completion of several distinct tasks. In the classical approach, the marking of all states that represent the completion of a task would imply, that all tasks have to be completed in one state and at the same time. From the practical point of view, this constitutes an unnecessary restriction of the DES modeling formalism as demonstrated in the following example.

A simple elevator only runs between two floors ($G_{uc,\, elevator}$ in Figure 1.1). There are four states describing the elevator being on floor one or two, or moving up or down, and four events `move_up`, `move_down`, and `up` and `down` which describe the arrival of the elevator at the respective floor. In the initial state, the elevator is on the lower floor. In another

model $G_{uc,\,doors}$, we consider the operation of the doors of the elevator which can be modeled by two states, one for doors open and the initial one for doors closed. The events consequently are `open_doors` and `close_doors`.

The specification $R_{uc,\,elevator}$ forbids the movement of the elevator when the doors are open. Furthermore, it guarantees that the doors do not open before the elevator has stopped.

As the supervisor should guarantee that the doors can be opened on every floor, we choose state 2 symbolizing open doors as marked state in $G_{uc,\,doors}$. We also want the elevator to move upwards and downwards. For this reason the corresponding states 2 and 4 of $G_{uc,\,elevator}$ are marked, too. As can be seen, these markings do not get along with our specification $R_{uc,\,elevator}$ and cause a conflict between states 2 and 4 of $G_{uc,\,elevator}$ and state 2 of $G_{uc,\,doors}$. The resulting supervisor $S_{uc,\,elevator}$ either enables the movement of the elevator what implies that the doors have to be closed, or it allows the doors to be opened which requires that the elevator has arrived at one of both floors and does not move until the doors are closed again. Hence, simultaneous completion of both tasks is not possible.

To solve this problem, simple markings have been replaced by colored markings in [dQC04], [dQC05]. So, the completion of several tasks can be marked by several types of markings, namely colors (Figure 1.2). A Colored Marking Supervisor as is introduced then assures that all tasks can be completed independently of each other. In accordance to our example, this means opening the doors and movement of the elevator do not have to happen at the same time, but can occur alternately, what resolves our conflict.

However, introducing colored markings does not solve the problem of state space explosion. To this end, K. Schmidt, M. H. Queiroz and J. E. R. Cury combined the idea of hierarchical control and colored marking and showed the efficiency of the combined approach [SQC07].

In this thesis, it is explained how we integrated a multitasking plugin into the libFAUDES software library. It offers the necessary methods for creating CMGs and for operating on them, for instance for inserting colors into states, for state minimization, the parallel composition or the strongly trim operations. Functions for the supervisor synthesis and abstraction also are available. Together, the examination of hierarchical multitasking DES and the synthesis of multitasking supervisors is enabled.

First, Chapter 2 describes basic facts concerning languages, generators and their properties. Furthermore, an introduction to CMGs and hierarchical design methods is given.

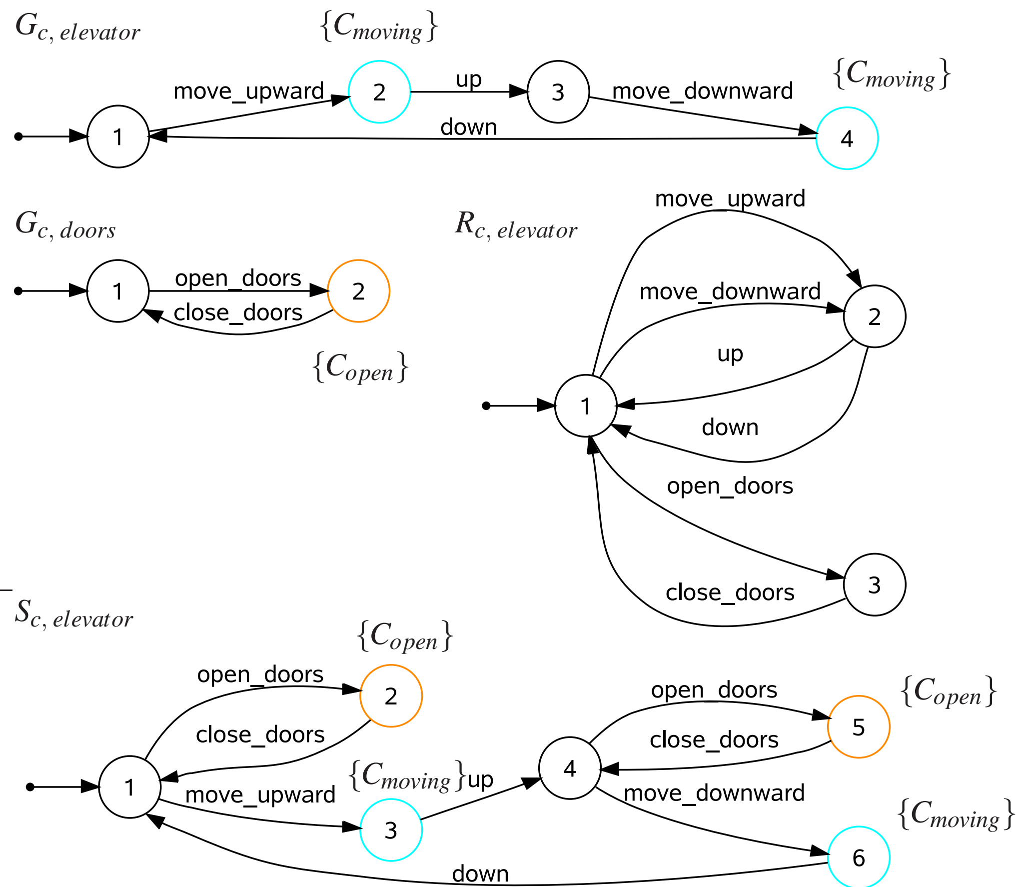


Figure 1.2: Model, specification and supervisor with colored states for a simple elevator

Chapter 3 offers detailed information about the libFAUDES multitasking plugin, especially about the data structure for saving color labels and the adaption of the required and already mentioned functions.

To conclude, two examples shall illustrate the effectiveness of hierarchical multitasking control of DES in Chapter 4. The first one is an adaption of the cat and mouse in a maze example introduced by Ramadge and Wonham [RW87], the second one realizes a part of a manufacturing system model K. Schmidt described in his dissertation [Sch05] and which is available at the Chair of Automatic Control.

# Chapter 2

# Definitions

At first, a short introduction to the topic of DES shall be given and basic definitions are presented. For further information and proofs, interested readers are advised to consult appropriate literature, e.g. [CL99] for general aspects of DES or [dQC04] regarding multitasking systems. Details about hierarchical and decentralized multitasking control, especially, can be found in [SQC07].

For illustrating some of the following definitions, a generator example as depicted in Figure 2.1 is introduced.

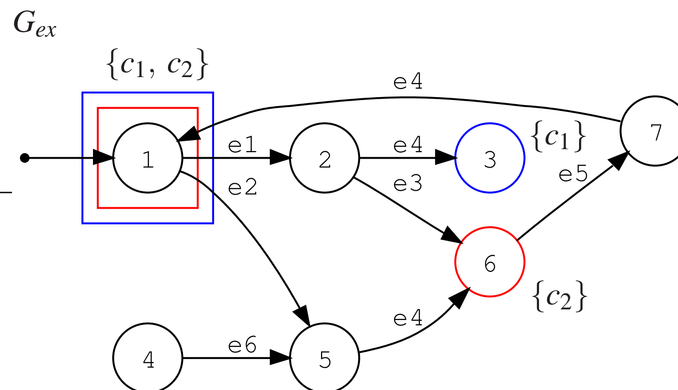


Figure 2.1: Automaton $G_{ex}$ for illustration of following definitions

In our automaton graphs, colored markings result in the coloring of the state node, if a state only contains one color label. In case of several colored markings, it is framed by rectangles in the appropriate colors. Besides, the respective color names are printed next to the corresponding states in curly brackets.

## 2.1 Formal Languages and Automata

Let $\Sigma$ be an event set (*alphabet*). The set containing the empty string $\varepsilon$ and all finite strings of elements of $\Sigma$ is denoted as the *Kleene-Closure* $\Sigma^*$. A language $L$ is a subset of $\Sigma^*$ and it is *prefix closed* if $L = \overline{L}$ with

$$\overline{L} := \{s \in \Sigma^* : \exists t \in \Sigma^* \text{ s.t. } (st \in L)\}$$

being the *prefix-closure*.

In this case, $\overline{L}$ contains all strings in $L$ and their prefixes, whereas in general, $L \subseteq \overline{L}$.

A *Colored Marking Generator (CMG)* is defined as a 6-tuple

$$G = (Q,\, \Sigma,\, C,\, \delta,\, \chi,\, q_0),$$

where $Q$ is a set of states, $\Sigma$ a set of events, $C$ a set of colors, $\delta : Q \times \Sigma \to Pwr(Q)$ (power set of $Q$) denotes the transition function, $\chi : Q \to Pwr(C)$ describes the marking function, and $q_0$ is the initial state.

$\Gamma : Q \to Pwr(\Sigma)$ is called *eligible event function* or *active event function* such that $\Gamma(q)$ is the set of events that are feasible in $q \in Q$.

For exemplifying this definition, we refer to the automaton $G_{ex}$ from Figure 2.1, which establishes the sets

$$\begin{aligned}
Q &= \{1,\, 2,\, 3,\, 4,\, 5,\, 6,\, 7\},\\
\Sigma &= \{e1,\, e2,\, e3,\, e4,\, e5,\, e6\},\\
C &= \{c_1,\, c_2\},\\
\delta(1,\, e1) = 2,\;\; \delta(2,\, e3) &= 6,\;\; \delta(1,\, e2) = 5,\; ...,\\
\chi\{1\} = \{c_1,\, c_2\},\;\; \chi(3) &= \{c_1\},\;\; \chi(6) = \{c_2\},\\
q_0 &= \{1\}, \text{ and}\\
\Gamma\{1\} = \{e1,\, e2\},\;\; \Gamma(2) &= \{e3,\, e4\},\;\; \Gamma(5) = \{e4\},\; ...
\end{aligned}$$

A CMG is termed *finite automaton* when its set of states is finite. Furthermore, it is called *deterministic* if it only contains one initial state $q_0$ and if in every state $q$ there are no active events $\gamma$ appearing multiple times and leading to separate successor states. Otherwise, it is nondeterministic.

The *language generated by the generator G* is designated as

$$L(G) := \left\{s \in \Sigma^* \mid f(q_0,\, s) \text{ is defined}\right\}.$$

As for multiple classes of tasks in one CMG different color labels $c \in C$ are used for markings, the languages

$$L_c(G) = \Big\{s \in L(G) \mid c \in \chi\big(\delta(q_0(s))\big)\Big\} \in Pwr(\Sigma^*), \forall c \in C$$

mark the completion of a task of a respective class or color $c \in C$. Consequently,

$$\Lambda_C = \big\{(L_c \mid c),\ c \in C\big\} \subseteq Pwr\left(Pwr(\Sigma^*) \times C\right)$$

defines a *colored behavior*, i.e., the set of pairs of colored languages with their respective color.

For a nonempty set of colors $B$ with $\emptyset \subset B \subseteq C$ the *language marked by* $B$ is defined as

$$L_B(G) := \big\{s \in L(G) \mid B \cap \chi\big(\delta(q_0,\ s)\big) \neq \emptyset\big\},$$

what means that any generated string $s$ is a prefix of any completed task marked with a color $b \in B$.

For our example automaton $G_{ex}$ depicted in Figure 2.1, the language generated by $G_{ex}$ results as

$$\begin{aligned} L(G_{ex}) = \big\{&e1,\ e1e4,\ e1e3e5,\ e1e3e5e4,\ e1e3e5e4e1,\ ..., \\ &e2,\ e2e4,\ e2e4e5,\ e2e4e5e4,\ ...\ \big\}, \end{aligned}$$

its colored behavior as

$$\begin{aligned} \Lambda_C(G_{ex}) = \Big\{&\big(\{e1e4,\ e1e3e5e4,\ e2e4e5e4,\ e1e3e5e4e1e4,\ ...\},\ c_1\big), \\ &\big(\{e1e3,\ e2e4,\ e1e3e5e4,\ e2e4e5e4,\ ...\},\ c_2\big)\Big\}. \end{aligned}$$

## 2.2 Natural Projection

There are two event sets $\Sigma_1$ and $\Sigma_2$, which naturally are subsets of $\Sigma_1 \cup \Sigma_2$. Then, the *natural projection* $P_i :\ (\Sigma_1 \cup \Sigma_2)^* \to \Sigma_i^*$ for $i = 1, 2$ erases events in a string formed from the larger event set $\Sigma_1 \cup \Sigma_2$, that do not belong to the smaller one ($\Sigma_1$ or $\Sigma_2$).

It is defined as follows:

$$P_i(\varepsilon) := \varepsilon,$$

$$P_i(e) := \begin{cases} e & \text{if } e \in \Sigma_i \\ \varepsilon & \text{if } e \notin \Sigma_i. \end{cases}$$

Strings can be viewed as the concatenation of single elements such that

$$P_i(se) := P_i(s)\, P_i(e)$$

for the prefix $s \in (\Sigma_1 \cup \Sigma_2)^*$ and an event $e \in (\Sigma_1 \cup \Sigma_2)$.

The projection for the language $L(G)$ can be obtained by applying it to all strings in the language.

With the natural projection $p_0 : \Sigma^* \to \Sigma_0^*$ the colored natural projection $m_0 : Pwr(Pwr(\Sigma^*) \times C) \to Pwr(Pwr(\Sigma_0^*) \times C)$ for CMGs is defined such that

$$L_c\left(m_0(\Lambda_C)\right) = p_0\left(L_c(\Lambda_C)\right), \;\; \forall c \in C.$$

The projected colored behavior can again be realized by a CMG $G_0$ such that for each $c \in C$, $\Lambda_C(G_0) = m_0(\Lambda_C(G))$.

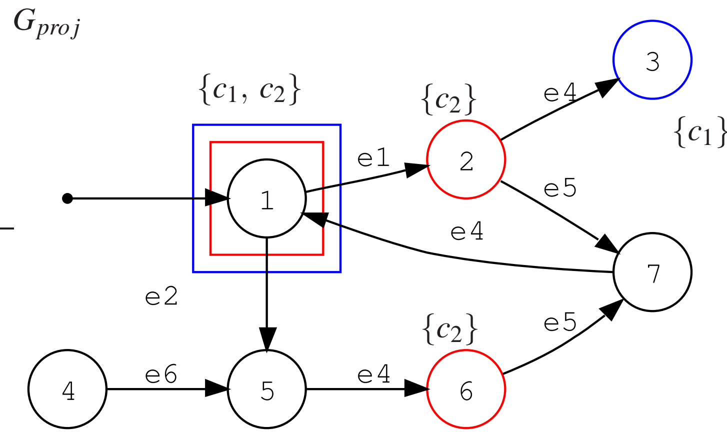


Figure 2.2: $G_{proj}$ as the projection's result for $G_{ex}$ and the alphabet $\Sigma_1 = \{e1, e2, e4, e5, e6\}$

The colored natural projection applied on the automaton $G_{ex}$ from Figure 2.1 with the alphabet $\Sigma_1 = \{e1, e2, e4, e5, e6\}$, where $e3$ is missing, consequently results in the automaton $G_{proj}$ shown in Figure 2.2.

The result for the language generated by $G_{proj}$

$$L(G_{proj}) = \{e1,\, e1e4,\, e1e5,\, e1e5e4,\, e1e5e4e1,\, \ldots \\ e2,\, e2e4,\, e2e4e5,\, e2e4e5e4,\, \ldots\}$$

and the colored behavior for $G_{proj}$

$$\Lambda_C(G_{proj}) = \{(\{e1e4,\, e1e5e4,\, e2e4e5e4,\, e1e5e4e1e4,\, \ldots\},\, c_1), \\ (\{e1,\, e2e4,\, e1e5e4,\, e2e4e5e4,\, \ldots\},\, c_2)\}$$

differ from $L(G_{ex})$ and $\Lambda_C(G_{ex})$ in that way, that in the respective strings contained in $L(G_{proj})$ and $\Lambda_C(G_{proj})$ the event $e3$ is erased.

The inverse projection $P_1^{-1} : \Sigma_1^* \rightarrow Pwr((\Sigma_1 \cup \Sigma_2)^*)$ extends a smaller event set $\Sigma_1$ by all events $e \in \Sigma_1 \cup \Sigma_2$ with $e \in \Sigma_2 - \Sigma_1$. In an automaton representation, the inserted events appear in all states as self-loops (see Figure 2.3).

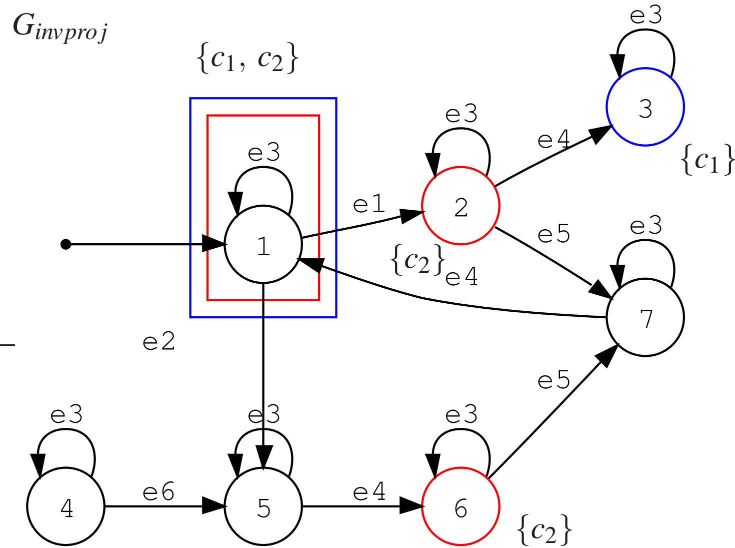


Figure 2.3: Inverse projection applied on the automaton $G_{proj}$, where event $e3$ was missing, with the original alphabet $\Sigma_1$ containing $e3$

## 2.3 Accessibility and Coaccessibility

Accessibility describes an automaton's property that all its states can be reached by starting in an initial state and following the transition relation. So, state $q \in Q$ is *accessible* if there exists an $s \in \Sigma^*$ such that $\delta(q_0, s) = q$. $G$ is called accessible if $q$ is accessible for every $q \in Q$.

A state $q \in Q$ of a CMG is *weakly coaccessible w.r.t. its color set* $C$ if starting at $q$ at least one state that marks one or more colors of $C$ can be reached with an existing sequence of transitions. This means, there exists a $c \in C$ and a $s \in \Sigma^*$ such that $c \in \chi(\delta(q, s))$.

$q \in Q$ is *strongly coaccessible w.r.t.* $C$ if starting at state $q$ there is a sequence of transitions for every $c \in C$ that leads to a state marked by $c$:

$$\forall c \in C, \exists s \in \Sigma^* \text{ s.t. } c \in \chi(\delta(q, s)).$$

A CMG is *weakly trim*, if it is accessible and weakly coaccessible w.r.t. $c$. It is called *strongly trim*, if it is accessible and strongly coaccessible.

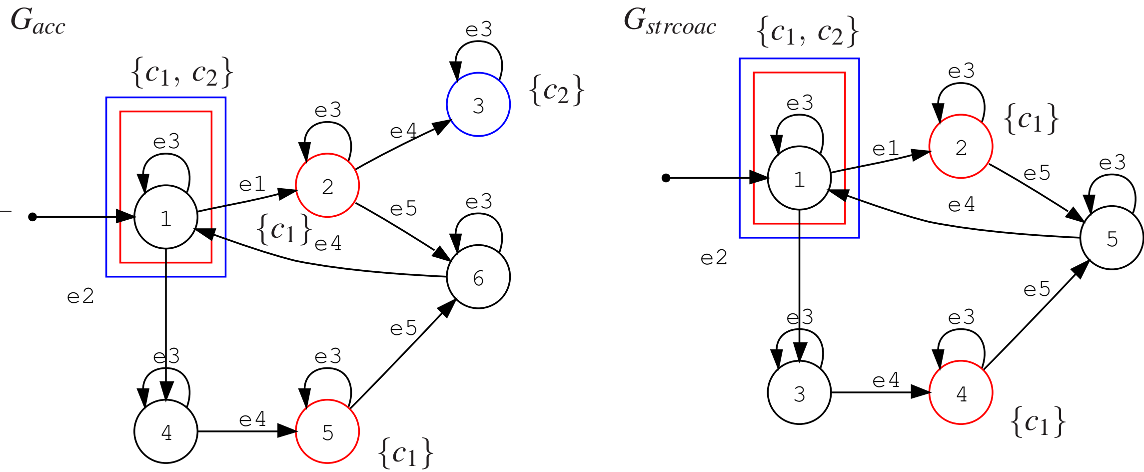


Figure 2.4: The accessible form of the automaton $G_{invproj}$ from Figure 2.3 is shown on the left side, its also strongly coaccessible version on the right.

The automaton $G_{invproj}$ from Figure 2.3 is already weakly coaccessible w.r.t. the color set $C = \{c_1, c_2\}$, as from any state at least one colored state can be reached. However, it is not accessible and not yet strongly coaccessible. When applying the accessible operation on the automaton, it looses state 4 (Figure 2.4). To even make it strongly coaccessible, state 3 has to be erased, as from there no state colored by $c_2$ can be reached.

## 2.4 Blocking

A CMG G is *weakly nonblocking w.r.t. C* if

$$L(G) = \overline{L_C(G)}.$$

So, any generated string can complete a task marked with a color $c \in C$. An equivalent statement is that the accessible part of $G$ is weakly coaccessible.

G is *strongly nonblocking w.r.t. C* if

$$\forall c \in C,\ L(G) = \overline{L_c(G)},$$

which says that any string can be completed to all tasks. Equivalently, the accessible part of $G$ has to be strongly coaccessible.

Using those definitions, we examine the blocking behavior of $G_{ex}$ from Figure 2.1 by regarding the appropriate languages. The language generated by $G_{ex}$ is

$$L(G_{ex}) = \{e1,\ e2,\ e1e3,\ e1e4,\ e2e4,\ e1e3e5,\ ...\},$$

whereas the prefix closures of the colored marked languages result to

$$L_{c_1}(G_{ex}) = \{e1e4,\ e1e3e5e4,\ e2e4e5e4,\ e2e3e5e4e1e3,\ ...\} \text{ and}$$
$$L_{c_2}(G_{ex}) = \{e1e3,\ e2e4,\ e1e3e5e4,\ e2e4e5e4,\ ...\}.$$

As $\overline{L_{c_2}}$ does not contain the prefix $e1e4$ which is contained in $L(G_{ex})$, the automaton cannot be strongly nonblocking. $\overline{L_{c_1}}$, however, is identical to $L(G_{ex})$. Therefore, at least weakly nonblocking is given.

## 2.5 Colored Parallel Composition

The parallel composition of two colored behaviors $M_B$ and $N_C$, with $M_B \subseteq N_C$ if $B \subseteq C$ and $\forall b \in B,\ L_b(M_B) \subseteq L_b(N_C)$, is defined as

$$\begin{aligned} M_B \parallel M_C := & \{L_b(M_B) \parallel L_b(N_C),\ b,\ \forall b \in B \cap C\} \cup \\ & \{(L_b(M_B) \parallel \overline{L_C(N_C)},\ b),\ \forall b \in B - C\} \cup \\ & \{(\overline{L_B(M_B)} \parallel L_b(N_C),\ b),\ \forall b \in C - B\} \end{aligned}$$

with a resulting color set $B \cup C$.

For exemplification, those colored behaviors can be represented as two automata with colored markings. With the definition above, their composition can be explained as follows.

A colored label existing in the current states of both automata to combine leads to the composition's resulting state also possessing this label. On the other hand, colors appearing in both automata, but only in one of the compared states are not considered in the respective composition state. However, if a state's color in one of the single automata does not appear in the other one at all, then it is adopted to the result.

Therefore, the parallel composition for two CMGs $G_1 = (Q_1,\ \Sigma_1,\ C_1,\ \delta_1,\ \chi_1,\ q_{0,1})$ and $G_2 = (Q_2,\ \Sigma_2,\ C_2,\ \delta_2,\ \chi_2,\ q_{0,2})$ is defined as

$$G_1 \parallel G_2 := Ac\big(Q_1 \times Q_2,\ \Sigma_1 \cup \Sigma_2,\ C_1 \cup C_2,\ \delta,\ \chi,\ (q_{0,1},\ q_{0,2})\big), \text{ with}$$

$$\delta\big((q_1,q_2),\,\sigma\big) = \begin{cases} \big(\delta_1(q_1,\sigma),\ \delta_2(q_2,\sigma)\big), \text{ if } \sigma \in \Gamma_1(q_1)\cap\Gamma_2(q_2) \\ \big(\delta_1(q_1,\sigma),\ q_2\big), \text{ if } \sigma \in \Gamma_1(q_1)\backslash\Sigma_2 \\ \big(q_1,\ \delta_2(q_2,\sigma)\big), \text{ if } \sigma \in \Gamma_2(q_2)\backslash\Sigma_1 \\ \text{undefined otherwise,} \end{cases}$$

$$\chi((q_1,q_2)) = [\chi_1(q_1)\cup(C_2-C_1)]\cap[\chi_2(q_2)\cup(C_1-C_2)],$$

$$\Gamma\big((q_1,q_2)\big) = [\Gamma_1(q_1)\cup(\Sigma_2-\Sigma_1)]\cap[\Gamma_2(q_2)\cup(\Sigma_1-\Sigma_2)],$$

and $Ac$ being the accessible operation which deletes unreachable states from a generator.

In general, $L(G_1 \parallel G_2) = L(G_1) \parallel L(G_2)$. Furthermore, if $C_1 = C_2$ or if $G_1$ and $G_2$ are weakly coaccessible w.r.t. their respective color sets, the relation $\Lambda_C(G_1 \parallel G_2) = \Lambda_C(G_1) \parallel \Lambda_C(G_2)$ is valid as well.

## 2.6 Multitasking Supervisory Control

An open loop behavior of a DES, which is modeled by a CMG, has to be controlled by a supervisor such that a safety specification $A_D$, the *admissible language*, is fulfilled. Moreover, strongly nonblocking of the controlled system has to be guaranteed.

A *coloring supervisor* $S : L(G) \rightarrow Pwr(\Sigma)\times Pwr(E)$ where $Pwr(\Sigma)$ represents the set of enabled events and $Pwr(E)$ a set of new colors which indicate the completion of a task is *admissible* if $\forall s \in L(G),\ \Sigma_u \cap \Gamma(\delta(q_0,\ s)) \subseteq \mathcal{R}(S(s))$.

Here, $\Sigma_u$ contains all uncontrollable events and $\mathcal{R}(S(s))$ identifies the events the supervisor does not prevent after occurrence of the string s. Thus, it always enables all uncontrollable events of the active event set.

An admissible coloring supervisor, which is strongly nonblocking w.r.t. $D$ such that $\Lambda_D(S/G) = A_D$ and $L(S/G) = \overline{L_D(A_D)}$, exists if the following conditions are fulfilled:

- controllability: $\overline{L_D(A_D)}\Sigma_u \cap L(G) \subseteq \overline{(L_D(A_D))}$
- D-closure: $L_d(A_D) = \overline{L_d(A_D)\cap L_d(G)},\ \forall d \in (D\cap C)$
- strong nonblocking of $A_D$ w.r.t. $D$

Controllability means that any uncontrollable event occurring after an allowed string that is feasible in the automaton's current state always has to be permitted by the supervisor, as disabling it is impossible anyway.

In case that controllability and strongly nonblocking are not fulfilled, the supremal controllable and strongly nonblocking supervisor $SCNB(A_D, G, D)$ can be computed with complexity polynomial in the number of states of the model [dQC04].

## 2.7 Multitasking Hierarchical and Decentralized Control

System models can often be composed by separate controlled DES $G_i$ with a respective color set $C_i$ which are represented by finite automata. The overall system model results as

$$G := \|_{i=1}^{n} G_i \text{ with color set } C := \bigcup_{i=1}^{n} C_i$$

$\Sigma_{i,u}$ and $\Sigma_{i,c}$ denote the uncontrollable and controllable events of the subsystem $G_i$. The events shared in different components are designated as $\Sigma_s$ and have to agree on their control status, what means that for two subsystems $G_i$ and $G_k$ with $i \neq k$

$$\Sigma_{i,u} \cap \Sigma_{k,c} = \emptyset.$$

If this condition is fulfilled, $\Sigma_u$ and $\Sigma_c$ of the overall system result as

$$\Sigma_u = \bigcup_{i=1}^{n} \Sigma_{i,u}$$
$$\Sigma_c = \bigcup_{i=1}^{n} \Sigma_{i,c}.$$

Before composing subsystems, they mostly can be reduced to a smaller number of states by natural projection. In this step, events of a component $G_i$ being not essential for the high-level are erased from the low-level alphabet. This procedure called *hierarchical abstraction* is based on the idea that the low-level supervisor takes care of all low-level events. All other ones - especially all shared events - are contained in the high-level alphabet $\Sigma_{0,i}$, such that $\Sigma_s \cap \Sigma_i \subseteq \Sigma_{0,i} \subseteq \Sigma_i$.

As follows, colored marking high-level plants $G_{0,i}$ for the respective subsystems $G_i$ result as

$$\begin{aligned} L(G_{0,i}) &= p_i\big(L(G_i)\big) \\ \Lambda_C(G_{0,i}) &= m_{0,i}\big(\Lambda_C(G_i)\big) \end{aligned}$$

with the natural projection $p_i : \Sigma_i^* \rightarrow \Sigma_{0,i}^*$ and the colored projection $m_{0,i} : \Sigma_i^* \rightarrow \Sigma_{0,i}^*$.

Then, the overall high-level plant evaluates to the CMG $G_0$ with

$$\begin{aligned} L(G_0) = &\quad p_0(\|_{i=1}^n L(G_i)) = \quad \|_{i=1}^n L(G_{0,i}) \\ \Lambda_C(G_0) = &\quad m_0(\|_{i=1}^n \Lambda_C(G_i)) = \quad \|_{i=1}^n \Lambda_C(G_{i,0}), \end{aligned}$$

with the controllability properties taken over from the low-level.

The specification needed for computing a high-level supervisor is given as a colored behavior $A_D$ with its color set $D$. Based on it, a supremal controllable and strongly nonblocking supervisor $S_0 : L(G_0) \to Pwr(\Sigma_0) \times Pwr(E)$ with a set of new colors $E = D - C$ can then be computed.

The control action for the low-level supervisor $S : L(G) \to Pwr(\Sigma) \times Pwr(E)$ is defined for each $s \in L(G)$ as

$$S(s) := \Big( S_0\big(p_0(s)\big) \cup (\Sigma - \Sigma_0), \mathcal{I}\left(S_0(p_0(s))\right) \Big)$$

with $\mathcal{I}\left(S_0(p_0(s))\right)$ representing the high-level supervisor's colors.

Thus, the low-level supervisor compasses an alphabet with all events from the high-level supervisor and, additionally, the low-level events that are not of interest to the high level and, therefore, are not in the abstraction alphabet. The colors used at the high level are the same as in the subsystems. They cannot be neglected, as they stand for tasks to complete.

The control action each subsystem can observe after a string $s$ can be described as

$$(\mathcal{R}\,(S(s)) \cap \Sigma_i,\ \mathcal{I}\,(S(s)) \cap (C_i \cup E)).$$

Following the mentioned steps, hierarchical consistency can be guaranteed and the supervisor implementation is carried out such that $p_0(L(S/G)) = L(S_0/G_0)$.

To ensure a nonblocking behavior of the supervised system, the observer condition for CMGs is required.

**Definition 2.7.1 (Colored Observer)** *Let $L = \overline{L} \subseteq \Sigma^*$ be a language, $\Lambda_C \in Pwr(Pwr(\Sigma^*) \times C)$ with $L_C(\Lambda_C) \subseteq L$ a colored behavior, and define the natural projection $p_0 : \Sigma^* \to \Sigma_0^*$ and the colored natural projection $m_0 : Pwr\big(Pwr(\Sigma^* \times C)\big) \to Pwr\big(Pwr(\Sigma_0^* \times C)\big)$ for $\Sigma_0 \subseteq \Sigma$. Then, $m_0$ is a $\Lambda_C$-observer (w.r.t. $L$) iff for each $c \in C$, it holds that $\forall s \in L$ and $\forall t \in \Sigma_0^*$ with $p_0(s)t \in p_0\big(L_c(\Lambda_C)\big)$*

$$\exists u \in \Sigma^* \text{ s.t. } su \in L_c(\Lambda_C) \wedge p_0(su) = p_0(s)t.$$

Strongly nonblocking control can be achieved for all $c \in C$ for $\overline{L_c(S/G)} = L(S/G)$ if $m_{0,i}$ is a $\Lambda_C(G_i)$-observer for $i = 1, \ldots, n$.

To demonstrate the meaning of this definition, we chose the following automaton, which is deduced from the workpiece detection element presented in Chapter 4.2.2. It shall be examined if the observer condition holds for two different projection alphabets, $\Sigma_1 = \{a,\, c,\, d,\, f\}$ and $\Sigma_2 = \{a,\, c,\, d,\, e,\, f\}$.

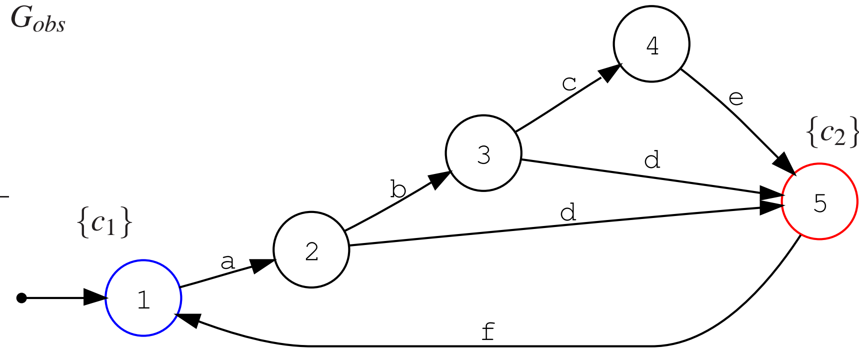


Figure 2.5: Automaton for illustrating the observer condition

$\Sigma_1 = \{a,\, d,\, e,\, f\}$: Let $p_1$ be the natural projection for the alphabet $\Sigma_1$. We examine the observer condition for $s = abc$ and $t = d$. As $p_1(G_{obs}) = \{ad,\, ae,\, adfad,\, \ldots\}$ and consequently $p_1(abc)d = ad \in p_1(G_{obs})$, $s$ and $t$ are valid values for testing the colored observer condition. With string $abc$ we arrive at state 4. However, there is no string $u$ such that $abcu \in L_{c_2}(G_{obs})$ and $p_0(u) = t$. So, it is shown for the color $c_2$ and the alphabet $\Sigma_1$ that $m_{\Sigma_1 \to \Sigma_{1,0}}$ is not a $\Lambda_C(G_{obs})$-observer.

$\Sigma_2 = \{a,\, c,\, d,\, e,\, f\}$: We regard the same string $s$ for the natural projection $p_2$ with the alphabet $\Sigma_2$. Other than in the previous case, the projection step now delivers the language $p_2(G_{obs}) = \{ad,\, ace,\, adfad,\, \ldots\}$. To attain the same critical case than above, we again choose $t = d$. Doing so, $p_2(s)t = p_2(abc)d = acd \notin p_2(G_{obs})$. Hence, this case does not affect the observer condition.

For all other strings $s$ and all other $t \in \Sigma_2$ the compliance with the observer condition can easily be verified.

**Definition 2.7.2 (Nonblocking Control)** *If $m_{0,i}$ is a $\Lambda_C(G_i)$-observer (w.r.t. $L(G_i)$) for $i = 1, \ldots, n$, then*

- *$m_0$ is a $\Lambda_C(G)$-observer (w.r.t. $L(G)$)*

- *the closed loop is nonblocking:* $\overline{L_c(S/G)} = L(S/G)$, $\forall c \in C$.

Note that the minimal generator for $p_0(L(G))$ maximally has as many states as the minimal generator for $L(G)$ [SQC07]. Moreover, the closed-loop on the high-level represents a finite automaton for which further abstraction may be possible.

# Chapter 3

# Multitasking Plugin for the libFAUDES Software Library

The libFAUDES software library developed at the Chair of Automatic Control at the Friedrich-Alexander-University Erlangen-Nürnberg is a tool for handling finite automata and regular languages. The most important feature in the supervisory control context is the possibility to compute minimally restrictive supervisors from a given plant model and an appropriate specification. The adequate algorithms and data structures for the classical approach [RW89] of marked generators have already been implemented. Also, algorithms for designing hierarchical systems as in [Sch05] for generators with a single color are available.

The libFAUDES project is based on an object-oriented concept developed in the programming language C++. Its sources are freely available under the terms of the GNU Lesser General Puclic License on our homepage [lib08] and may be used for own projects or be extended with own algorithms.

As was pointed out in the introduction of this thesis, the application of the classical approach is not always satisfactory for the supervisor design in multitasking systems. For this reason, we developed a libFAUDES plugin for the computational procedures for multitasking systems. This plugin introduces colored markings as state attributes that are used for functions such as the natural projection, the parallel composition or testing the observer condition presented in the previous section.

## 3.1 Plugin Description

The multitasking plugin consists of several files, which contain important functions and necessary classes for the handling of multitasking generators. Thereby, the class *TmtcGenerator* is the most important one for the software user. It offers methods for

- constructing and destructing generators,
- inserting and manipulating states, colored markings, events and transitions,
- naming and querying of data,
- input and output of generators or its single components, e.g. its color set.

Other functions, for instance the computation of the parallel composition or a colored supervisor and making a generator deterministic or accessible, are added in own files and do not belong to the *TmtcGenerator* class. Nevertheless, they belong to the plugin and are designed for the use with multitasking generators in Section 3.4.

The class *TmtcGenerator* inherits from *vGenerator* (Figure 3.1), which only contains virtual classes for the generator interface definitions. The derived class *TaGenerator* implements almost all methods for setting up generators in the classical approach. Controllability properties are realized in the class *TcGenerator*. As our multitasking plugin inherits from the *TcGenerator* class, many methods from the classes *TaGenerator* and *TcGenerator* can directly be used. Only methods which have to observe colored markings are reimplemented in *TmtcGenerator*.

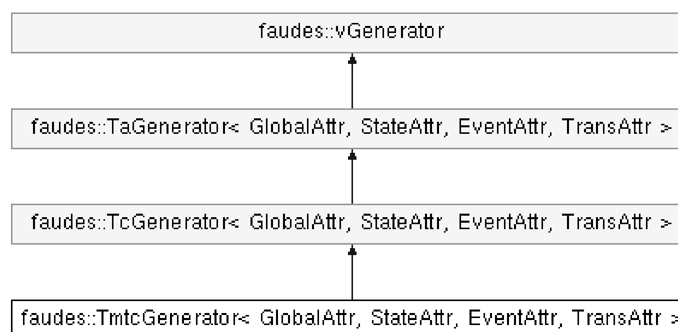


Figure 3.1: Inheritance diagram for class TmtcGenerator

The *TmtcGenerator* class is, as well as its base classes, realized as a template class, what is indicated by the preceding template parameter T in its identifier. Templates offer a very efficient way of defining properties, as only required parts are linked to the program.

Besides, the integration of templates is carried out during compilation and modifying them is possible with low complexity.

The compiler expects four class template parameters to be passed. They are

- *GlobalAttribute* for the generator,
- *StateAttribute* for single states,
- *EventAttribute* for single events, and
- *TransAttribute* for single transitions.

With colored markings affecting the states, they consequently are represented as state attributes, whereas the controllability properties of events are saved in the event attribute. The global attributes or transition ones are not needed for our approach. Thus, the void attribute *AttributeVoid* is inserted for them. However, they are already considered in the libFAUDES concept as a future implementation may require them.

The identifier of a class *TmtcGenerator<AttributeVoid, AttributeColoredState, AttributeCFlags,AttributeVoid>* is abbreviated as mtcGenerator by an appropriate C++ type declaration in our implementation. An example is given in Section 3.3.1.

## 3.2 Representation of Colors

The colors belonging to a particular state are saved in the class *AttributeColoredState*. It is inserted for every state as its corresponding state attribute StateAttr. The correlation with other classes and the respective data members are shown in the collaboration diagram in Figure 3.2.

The class *AttributeColoredState* is derived from *AttributeFlags* which in turn inherits from *AttributeVoid*. As we do not use any methods or data members from *AttributeFlags*, directly inheriting from *AttributeVoid* would also be possible, but for consistency with the class *AttributeCFlags* which takes care of the controllability status of events, *AttributeFlags* is taken as base class. Its member *mFlags* does not allocate any further cache memory, as its value is initialized to the static default value *mDefFlags*$= 0x0$. For this reason, the method *IsDefault* which only tests if *mFlags* = *mDefFlags*, is reimplemented in *AttributeColoredState* and then allows to easily find out if a state is colored or not. For that, the method additionally determines if *mColors* is empty.

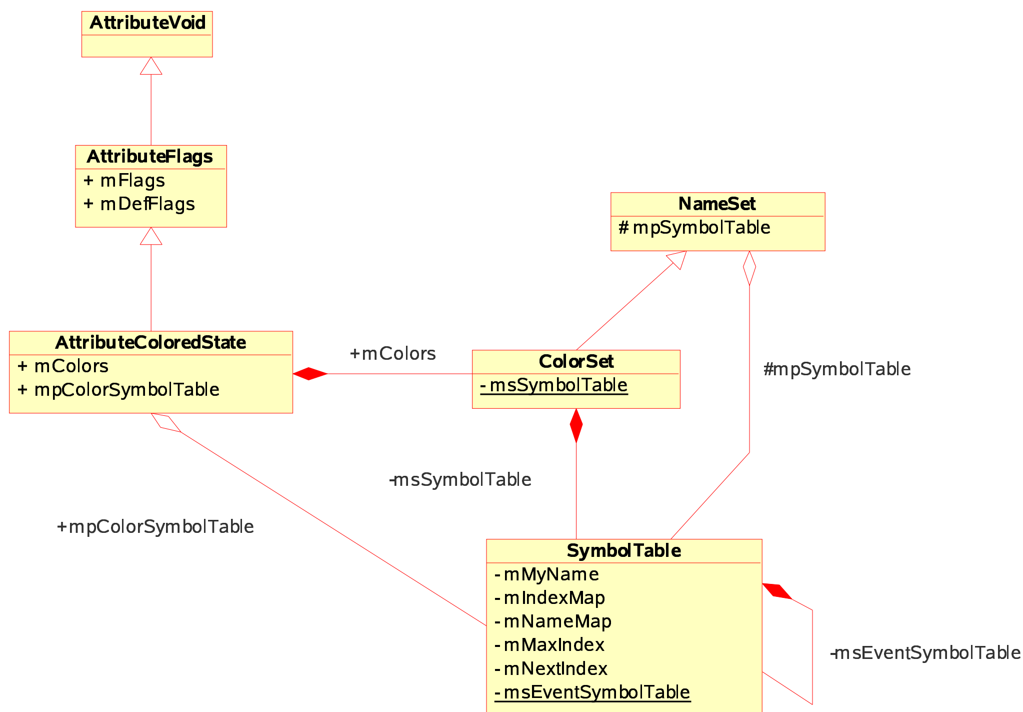


Figure 3.2: Class diagram for collaborating classes of AttributeColoredState

Each state's *AttributeColoredState* class has a color set mColors as data member, where the color labels eventually are saved. It is derived from *NameSet*, which is based on an index set *mIndexSet*. There, the color indices are inserted. *ColorSet* itself contains a static color symbol table, which means there is one color name symbol table for all color sets belonging to this generator. The pointer *mpSymbolTable* in class *NameSet* has to be set to this symbol table with every constructor call. Dereferencing it then is basis for saving color names and for looking them up again.

Color names usually are saved in a global color symbol table, which means there is one symbol table for all color sets in all generators. The member *mpSymbolTable* contained in *ColorSet* and *mpColorSymbolTable* from class *TmtcGenerator* are initialized to reference this global color symbol table.

Furthermore, the current implementation of *TmtcGenerator* also allows to set up a local color symbol table for each generator. Doing so, we are able to erase color names when deleting a color label from a generator's state. Using the global color symbol table what is the standard case, requires that color names are not erased, as it cannot be tested if there is another generator using the respective color as well. By contrast, establishing a local color symbol table resets the pointer *mpColorSymbolTable* accordingly, and consequently the comparison of this pointer with *mpSymbolTable* from class *ColorSet* tells us if a global

color symbol table or a local one is in use. In the latter case, the appropriate color name can be deleted.

Although using a local color symbol table has slightly been tested, its faultless functionality cannot be guaranteed. If this part of our plugin shall be used, it is in the programmers' hands to ensure accuracy.

## 3.3 Class “mtcGenerator”

### 3.3.1 Interface Methods

Multitasking generators are established using their abbreviated identifier *mtcGenerator*. The automaton $G_{ex}$ from Figure 2.1, for instance, can be generated using the following code sequence:

```
1   mtcGenerator gen;
2
3   st1 = gen.InsInitState();
4   st2 = gen.InsState();
5   st3 = gen.InsState();
6
7   eva = gen.InsEvent("a");
8   evb = gen.InsEvent("b");
9   evc = gen.InsEvent("c");
10
11  gen.SetTransition(st1, eva, st2);
12  gen.SetTransition(st1, evc, st3);
13  gen.SetTransition(st2, evb, st1);
14
15  gen.InsColor(st1, "init");
16
17  gen.Write("example.gen");
18  gen.ColorDotWrite("example.dot");
```

The first command establishes the generator *gen*. As is already known from the standard implementation, states, events and transitions are inserted next. The method *InsColor* then assigns state `st1` the color label *init*. The output methods *Write* and *ColorDotWrite*

then generate files with the specified names containing the generator information. *Write* thereby creates a .gen file, which is the standard libFAUDES generator file format. *ColorDotWrite* delivers a file in the Graphviz [gra08] .dot file format. It can be used to directly create a graphical output as an automaton.

In the following, some important methods provided by this class are presented. A detailed description is provided in the Doxygen [dox08] documentation.

- *InsColoredState* and *InsColor* are used to insert colored states or to add colored markings to existing states.
- *DelColor* erases a color from one or all states, *ClrStateColors* deletes all colors from one state whereas *ClrStateAttributes* deletes all colors from all states. *DelStates* erases whole states including their attributes.
- *Colors* returns all the generator’s colors, *StateColors* the ones of a respective state.
- *ColoredStates* delivers a state set containing all labeled states or states labeled with an appropriate color, respectively.
- The methods *ExistsColor* and *IsColored* can be used to find out about the color label status of a generator or a state.
- *ColorName* allows to find out the color name to an index, *ColorIndex* serves for the reverse case.
- *Accessible*, which is inherited from the base class, *StronglyCoac* and *StronglyTrim* are used to make generators reachable and strongly coreachable. *IsAccessible*, *IsStronglyCoac*, and *IsStronglyTrim* are used for testing those properties.

### 3.3.2 Implementation Details

#### Saving and Querying of Color Labels

The internal way of saving colors for a certain state or testing for its existence is always the same and shall be explained taking the following code sequence as an example.

```
1   StateAttr* attr = Attributep(StateIndex);
2   attr->mColors.Insert(ColorIndex);
```

The method *Attributep* returns a *StateAttr** pointer to the attribute specified by *StateIndex*. If there is not an attribute yet, a void attribute will be created and returned. The pointer then enables the access to the attribute’s according color set and its methods, for instance, for inserting a color.

If the color set belonging to a state shall only be examined, but not changed, the method *Attribute* which returns a constant reference to the according attribute should be used. Doing so, unintentional changes of the attribute can be avoided.

### Accessibility Operations

The *Accessible* operation, which ensures that all states in a generator are reachable, is reimplemented in *TmtcGenerator*, although accessibility does not depend on colored markings. It is necessary, because the base class implementation does not take care of possible attributes when removing unreachable states. The reimplemented method, by contrast, deletes both the forbidden states and their attributes. *IsAccessible* could be inherited from the base class, but for consistency, a wrapper function is implemented in *TmtcGenerator*.

The *StronglyCoac* and methods which address the stronlgy coaccessibility, have to be implemented in TmtcGenerator, as the color status is the crucial property to observe. Both operations are based on the method *StronglyCoacSet* which returns a state set with all strongly coaccessible states. As this function would affect the original generator by inserting marked states, a copy of the generator to examine is set up and *StronglyCoacSet* is applied on the copy. Therefore, the original generator does not need to be modified for finding the strongly coaccessible states and so *IsStronglyCoac* can be called on constant generators.

The functionality of the method *StronglyCoacSet* is as follows. First, it iterates over all the generator’s colors $c_i$. All states labeled with the current color $c_i$ then are set as marked states in the classical way, such that *CoaccessibleSet* from the base class can be carried out. This method returns the set of coaccessible states $Q_{tmp}$ relating to the classical markings - what in this case means - to the current color $c_i$. The intersection of all those coaccessible state sets for all colors $c_i$ finally results in the strongly coaccessible state set $Q_{strcoac}$ which is returned.

```
1  StateSet StronglyCoacSet(void) {
2    StateSet Q_strcoac;
3    for all colors c_i of generator G {
```

```
4      clear marked states;
5      set all states labeled with c_i as marked states;
6      // perform coaccessible operation from TaGenerator
7      StateSet Q_tmp = CoaccessibleSet();
8      Q_strcoac = Q_strcoac * Q_tmp;
9    }
10   clear marked states;
11   return Q_strcoac;
12 }
```

*StronglyCoac* uses this result and deletes all states being not strongly coaccessible. Comparing the strongly coaccessible state set with the generator's state set in the method *IsStronglyCoac* allows an assertion about the generator being strongly coaccessible or not.

## 3.4 Further Functions

Further important functions related to the supervisory control are implemented in separate files. This concerns

- functions such as *Deterministic*, *Project*, and *Parallel*, and
- the strongly nonblocking supervisor computation in *SupConNB*.

Their realization shall be explained in the following sections. Most of the particular functions' implementations take more parameters than absolutely necessary. That is why wrapper functions which only require necessary parameters and represent the user interface are established.

### 3.4.1 Deterministic

The function *Deterministic* (see Section 2.1) takes a nondeterministic mtcGenerator $G$ and creates a deterministic generator that has the same closed and colored languages as the original one. This resulting generator is inserted into the empty mtcGenerator $G_{det}$.

```
1 void Deterministic( const mtcGenerator& G,
2                     ...,
```

```
3                       mtcGenerator&          G_det  )
```

At first, the function checks if there exists at least one initial state $q_0$. If there is none, the function returns. Otherwise, one initial state $q_{0,det}$ is inserted into $G_{det}$.

In the classical theory, one of the original initial states being marked would lead to the resulting generator's initial state being marked, too. Concerning multitasking automata, this rule is adapted in that way that $G_{det}$'s initial state gets all color labels of all initial states of the original generator $G$. Therefore, for each initial state $q_{0,i}$ a set $C_i$ with its respective colors is generated and assigned to $G_{det}$'s initial state $q_{0,det}$.

```
1  q_0,det = G_det.InsInitState();
2  for all q_0,i in G {
3     ColorSet C = StateColors(q_0,i);
4     if (C ≠ ∅) → G_det.InsColor(q_0,det, C);
5  }
```

To determine all other states to insert in $G_{det}$, we start at $G$'s initial states and follow all feasible strings. All states which could be reached when following a particular string are combined to separate state sets $Q_i$. For each of those state sets, a state $q_{i,det}$ is inserted into $G_{det}$. Following the classical theory, a marked state contained in $Q_i$ would lead to the equivalent state $q_{i,det}$ also having to be marked. Related to multitasking automata this means, that all colors appearing in $Q_i$ have to be present in $q_{i,det}$, too. Therefore, an iteration over all states in $Q_i$ is started and all single states' color sets are added to $q_{i,det}$.

```
1  q_i,det = G_det.InsState();
2  for all states q in Q_i {
3     ColorSet C_i = G.StateColors(q_i);
4     if (C_i ≠ ∅) → G_det.InsColor(q_i,det, C_i);
5  }
```

### 3.4.2 Project

Within the Project operation (see Section 2.2), there also are states to combine. All states $q_{i,reach}$ which, starting from $q$, can be reached by transitions with events not being in the high-level alphabet $\Sigma_{proj}$, are merged into a single state $q_{i,proj}$ in the resulting automaton $G_{proj}$. If a single $q_{i,reach}$ is marked, then $q_{i,proj}$ also has to be marked. According to colored labeling, the equivalent solution is that all colors appearing in the respective original states

$q_{i,reach}$ have to be transferred to $q_{i,proj}$. This procedure is again carried out by iterating over $G$'s affected states $q_{i,reach}$ and inserting the appropriate color sets $C_i$ into $q_{i,proj}$.

```
1  void ProjectNonDet( mtcGenerator&    G,
2                            const EventSet&  Σproj  ){
3    ...
4    for all local accessible states qi,reach from q {
5      ...
6      ColorSet Ci = G.StateColors(qi,reach);
7      if (Ci ≠ ∅) → G.InsColor(qi,proj , Ci);
8    }
9  ...
10 }
```

### 3.4.3 Parallel

The parallel composition (see Section 2.5) in the classical theory handles two states $q_1$ and $q_2$ such that the new arising state $q_{1,2}$ is marked, if both original states were marked. However, if only one state is marked, but the regarded state from the other automaton does not have any marked state, then the composed system's equivalent state has to be marked, too. In all other cases, the new state stays unmarked.

Regarding colored markings, the procedure is very much the same. All colors, which appear in $q_1$ and $q_2$ have to be inserted to $q_{1,2}$ as well. All colors being part of one state $q_1$ or $q_2$ and which do not appear in the respective other generator at all, also have to be inserted to $q_{1,2}$.

In order to easier compute the parallel composition for generators with colored markings, a helper function called *ComposedColorSet* was implemented to determine the resulting state's color labels.

As parameters, both original generators $G_1$ and $G_2$ are passed together with their current states $q_1$ and $q_2$. Their particular color sets $C_1$ and $C_2$ are also given as parameters, although it would be possible to compute them in the function itself. In case of multiple usage as in the *Parallel* function, however, it is more efficient to generate these color sets once and then pass them to the particular subfunctions.

Eventually, a reference to a color set where to save the composed colors has to be passed. These colors can afterwards be inserted to the new state of the composed system.

The function is realized as follows.

```
1  void ComposedColorSet(
2              const mtcGenerator& G1,
3              const Idx q1, ColorSet& C1,
4              const mtcGenerator& G2,
5              const Idx q2, ColorSet& C2,
6              ColorSet& Ccomposed ) {
7
8    AttributeColoredState attr1, attr2;
9    attr1 = G1.States().Attribute(q1);
10   attr2 = G1.States().Attribute(q2);
11
12   if q1 is colored {
13     ∀ci ∈ Cq1 do {
14       if (ci ∈ C2) {
15         if q2 is colored and (ci ∈ Cq2) → Ccomposed.Insert(ci);
16       }
17       else → Ccomposed.Insert(ci);
18     }
19   }
20
21   if q2 is colored {
22     ∀cj ∈ Cq2 do {
23       if (cj ∉ C1) → Ccomposed.Insert(ci);
24     }
25   }
26 }
```

At first, both original generator's current states' attributes are obtained. They are necessary for accessing the color sets of the respective states.

Then, an iteration over all colors contained in state $q_1$ is started. If a particular color $c_i$ appears in generator 1 and generator 2, it is only taken over to the resulting generator if it is also contained in the second generator's current state. The condition if $q_2$ is colored or not from line 15, can be checked with low computational costs. That is why it is tested before all colors are gone through in order to find $c_i$.

A color only appearing in the first generator is directly inserted to the composed color set.

In the next step, the second iteration is carried out over all colors of $q_2$. Colors appearing in both generators have either been considered when examining $q_1$'s colors or they are irrelevant, because of not being part of $q_1$'s labels. So, only those colors solely existing in generator 2 have to be regarded. Their subset which is contained in $q_2$ consequently has to be added to the composed set.

### 3.4.4 SupConNB

The *SupConNB* function implements the computation of the supremal controllable and strongly nonblocking (SCSNB) supervisor (see Section 2.6) from a given model and an appropriate specification. In fact, it is only a wrapper function for *SupconParallel*, where the implementation is actually realized.

```
1  void SupconParallel( const mtcGenerator& G_plant ,
2                       const mtcGenerator& G_spec ,
3                       ... ,
4                       mtcGenerator&       G_res  )
```

The rules for markings in the classical way and for CMGs thereby are the same as described in the section parallel (see Section 3.4.3). Only colors appearing either in both generators' current states or appearing only in one regarded state and not being part of the other generator are inserted to the respective resulting generator's state. Consequently, the color labeling of particular states is carried out in the same way than in Parallel, what means that the already familiar function *ComposedColorSet* can be used again.

In the first step, both generators' color sets $C_{plant}$ and $C_{spec}$ are established. They comprise all colors occurring in the plant or the specification. Then, starting from their initial states $q_{0,plant}$ and $q_{0,spec}$ we begin with finding out about their color labels. For that step *ComposedColorSet* is called with the plant's and the specification's respective properties as parameters. The resulting composed color set $C_{composed}$ is then added to the new inserted initial state in the SCSNB generator.

```
1    ColorSet C_plant = G_plant.Colors();
2    ColorSet C_spec = G_spec.Colors();
3
4    ColorSet C_composed;
5    ComposedColorSet( G_plant , q_0,plant , C_plant ,
```

```
6                                 G_spec , q_0,spec , C_spec ,
7                                 C_composed );
8   if (C_composed ≠ ∅) {
9     Idx q_new = G_res.InsInitState();
10    ...
11    G_res.InsColor(q_new, C_composed);
12  }
```

From the model's and the specification's current states we then proceed by following common feasible transitions. For each new pair of states $(q_{plant}, q_{spec})$ reached by this method, we insert an appropriate state to $G_{res}$, compose its respective color set and set it as the new state's color set.

```
1   if (q_plant, q_spec) is new {
2     ComposedColorSet( G_plant , q_plant , C_plant ,
3                       G_spec , q_spec , C_spec ,
4                       C_composed );
5     if (C_composed ≠ ∅) {
6       Idx q_new = G_res.InsState();
7       G_res.InsColor(q_new, C_composed);
8     }
9     ...
10  }
```

Finally, all states forbidden by the supervisor, but already contained in $G_{res}$, are deleted after having erased their color labels.

```
1   for all forbidden states q_forbidden {
2     G_res.ClrStateAttribute(q_forbidden);
3   }
4   G_res.DelStates(Q_forbidden);
```

### 3.4.5 Statemin

The state minimization tries to reduce an automaton's number of states by merging states without changing the generator's language $L$ and - in the classical theory - the marked language $L_m$ generated by the generator. Accordingly, colored behaviors $\Lambda_C$ shall not be changed when state minimization is applied to CMGs.

In the classical theory, only states with the same marking status can be equivalent. This means, either all those states to merge are marked, or none of them. Relating to CMGs, an equivalent solution is that states can only be equivalent, if they possess the same colored markings. That is, all colors appearing in one state also have to be contained in the other states which should be merged with it, but no further ones.

The implementation of the state minimization algorithm requires two generators as parameters. The first one is the one to minimize, the second one holds the resulting generator.

```
1 void StateMin( mtcGenerator& rGen,
2               mtcGenerator& rResGen, ... )
```

Within the function, all distinct color sets $C_j$ of all generator's states $q_i$ are established. To every color set $C_j$ a set of those states which are labeled by it is assigned. What results is saved in the map $\langle C_j, Q_j \rangle$ which contains all possible color sets $C_j$ and the particular states $Q_j$ where they occur.

The vector $\langle Q_k \rangle$ also receives all state sets with states labeled by the same colors.

First, all uncolored states $q_{u,i}$ are inserted to a state set $Q_u$, which is added to $\langle Q_k \rangle$ at position $k = 0$.

```
1   if there are uncolored states q_{u,i} {
2     loop over all q_{u,i} and add them to Q_u;
3     add Q_u to vector ⟨Q_k⟩;
4     increase k;
5   }
```

Afterwards, an iteration over all colored states is started. All state color sets which are not yet contained in $\langle C_j, Q_j \rangle$, thereby are inserted into it. Finally, the states are inserted relating to their color labels to the corresponding state set.

```
1   for all states q_i {
2     if current state's color set C_i ∈ ⟨C_j, Q_j⟩ {
3       insert q_i into Q_j
4     }
5     else {
6       create new state set Q_new and insert q_j;
7       insert {C_j, Q_new} into ⟨C_j, Q_j⟩;
8     }
```

```
9   }
```

After these sorting procedures, all state sets $Q_j$ are added to the vector $\langle Q_k \rangle$. At the end, a loop over $\langle Q_k \rangle$ is carried out. The color labels for each state in the respective state set is detected by examining the first state. If it is nonempty, it is set as the color marking of the resulting generator's equivalent state.

```
1   for all state sets Q_j of ⟨C_j,Q_j⟩ {
2     add Q_j to ⟨Q_k⟩;
3     increase k;
4   }
5   ...
6   loop over all blocks ⟨Q_k⟩ {
7     get color set C_{k,1} of first state in ⟨Q_k⟩;
8     if (C_{k,1} ≠ ∅) → set C_{k,1} for equivalent state q_new in G_res;
9   }
```

### 3.4.6 UniqueInit

The function *UniqueInit* checks if there are multiple initial states. If so, they are combined to a new initial state $q_{0,unique}$ getting all color labels the original initial states $q_{0,i}$ possessed. The implementation of the respective mechanism is as follows. The initial states' single color sets are read and, provided that they are not empty, inserted into the new initial state's color set.

```
1 void UniqueInit( mtcGenerator& G ) {
2   for all q_{0,i} {
3     ...
4     ColorSet C = G.StateColors(q_{0,i});
5     if (C ≠ ∅) {
6       G.InsColor(q_{0,unique}, C);
7     }
8   ...
9 }
```

# Chapter 4

# Examples

In this chapter, two examples which have been evaluated using the libFAUDES multitasking plugin are presented. The first one is an adaption of the cat and mouse problem presented in [RW89] that was modified by Cury [dQC04]. It exemplifies the use of colored marking generators and the possibility to split up models for easier computation.

The second example describes a part of the Fischertechnik plant, a production plant model at the Chair of Automatic Control at the University Erlangen. This example is adequate to demonstrate the efficiency of multitasking hierarchical control and its algorithms.

## 4.1 Cat and Mouse in a Maze

In the cat and mouse in a maze example, the application of modular multitasking control is demonstrated. To this end, a model that consists of several components is controlled by disjoint supervisors instead of one monolithic supervisor. An advantage of doing so is the fact that the number of supervisor states is reduced. In combination with colored marking supervisors, it can also be assured that the completion of several independent tasks can be achieved.

### 4.1.1 Description of the Example and the Requirements

A maze with two identical floors and five rooms on each floor is investigated. The floor plan is shown in Figure 4.1. At the beginning, there is a cat in room 0 on the first floor. A mouse is situated in room 4 on the second one. For both animals there is some food in room 3 on floor 1.

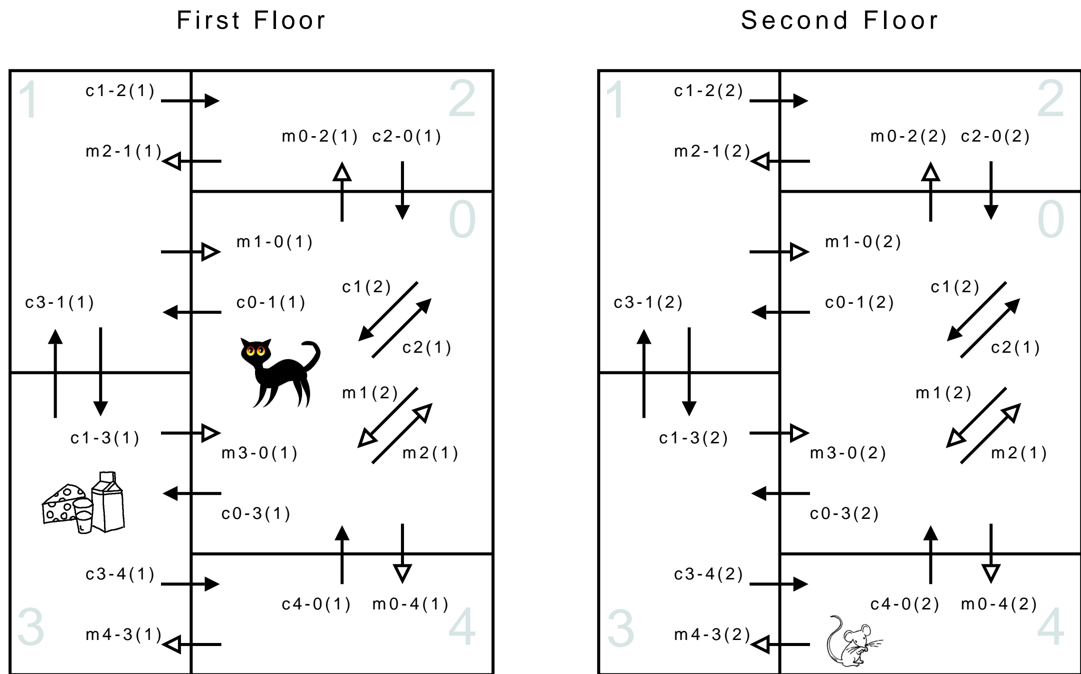


Figure 4.1: Both floors of the cat and mouse in a maze example

Cat and mouse cannot move freely from room to room. The possible paths for each animal are shown in the figure above. Each event `mx-y(i)` indicates that the mouse moves from room x to room y on level i of the maze. An event `cx-y(i)` accordingly denotes the movement from room x to room y on level i for the cat.

As there is a connection between both floors in each level's room 0, floor changes are possible for both animals. `mu(v)` and `cu(v)` specify the mouse and the cat, respectively, moving from room 0 on floor v to room 0 on floor u.

All room changes except the uncontrollable events $\Sigma_{uc}$ = {`c3-1(1), c1-3(1), c3-1(2), c1-3(2)`} which specify the cat going from room 3 to room 1 or vice versa can be prevented, as they represent controllable events.

Preventing one event in this case can be imagined as a door which can be opened and closed by a supervisor. For the four uncontrollable events there would be no doors to close, so on both levels the cat can move from room 3 to room 1 or vice versa without the supervisor being able to forbid it.

The task of the supervisor is to avoid that cat and mouse stay in the same room. Furthermore, it has to guarantee that both animals can move with maximal freedom, that returning to their respective initial rooms is possible, and that they always have a chance to access the food.

### 4.1.2 Modeling and Specification

For modeling the plant, room models for each room on both floors and for both animals were created. This resulted in twenty room models for five rooms on two floors and two animals. An automaton $C_r^l$ denotes the model for the cat in room r on floor l. As an example, the model automaton $C_0^1$ for room 0 on floor 1 is shown in Figure 4.2. All other rooms are created in the same way.

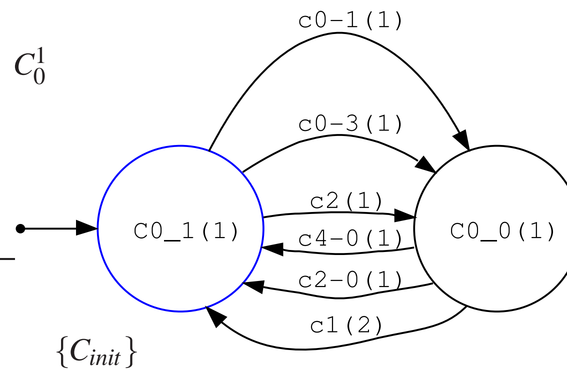


Figure 4.2: Model $C_0^1$ for room 0 on floor 1 for the cat.

A state *Cx_y(i)* thereby represents the number of cats (y) being in room x on floor i. In this example, the colored state *C0_1(1)* expresses that the cat should always be able to return to the initial state of room 0 on floor 1. The other one represents the empty room.

In addition to the single room models, there is a counter model for each floor (Figure 4.3). It observes the fact that there only is one cat. It is necessary, because otherwise the parallel composition would generate states for more than one cat in the system, what is excluded by the example definition.

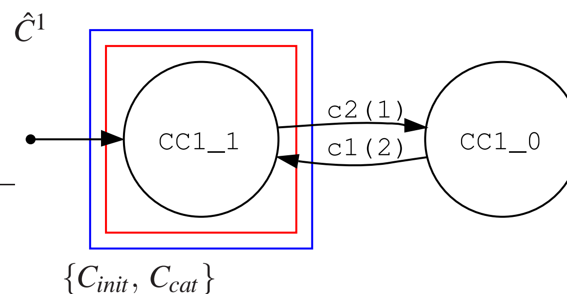


Figure 4.3: Counter model $\hat{C}^1$ for the cat on floor 1

A supervisor now shall assure that it is always possible for the cat and the mouse to return to their initial states, what implements that the cat can always come back to room 0 on floor 1 and the mouse to room 4 on floor 2. Recall that this requirement is captured by labeling the respective states *C0_1(1)* and *M4_1(2)* with the color $C_{init}$.

Other states to be marked with separate colors are *C3_1(1)* which is marked by color $C_{cat}$ and *M3_1(1)* which is marked by color $C_{mouse}$. These states describe the states where the cat or the mouse, respectively, are eating independently from each other.

Sticking to the classical theory with one sort of marked states, independent attainability of all marked states could not be ensured. In fact, the state where the cat is eating and the one where it is in its initial state can impossibly be assigned to one single marked state, as the cat cannot be in two rooms at the same time. Moreover, simply marking multiple states would mean that it does not matter which of the states has to be reached. However, in our case we want the cat to both come back to the initial state and be able to eat. What follows is that the different meanings of markings go lost when composing the system and thus, the supervisor cannot guarantee the possibility to reach all originally marked states.

With the introduction of colored markings and the appropriate colors $C_{cat}$, $C_{mouse}$, and $C_{init}$, by contrast, a multitasking supervisor is able to guarantee the independent reachability of these three aims from all the system's states.

The specification also is divided into several parts, one specification automaton for every room. One example for room 0 on floor 1 is shown in Figure 4.4. It represents a counter with three states: *c0_0(1)* denotes that the room is empty, *c0_1(1)* indicates that the cat is inside and *m0_1(1)* stands for the mouse being present. So, this specification automaton prevents the cat and the mouse being in one room at the same time, as both animals only can enter empty rooms.

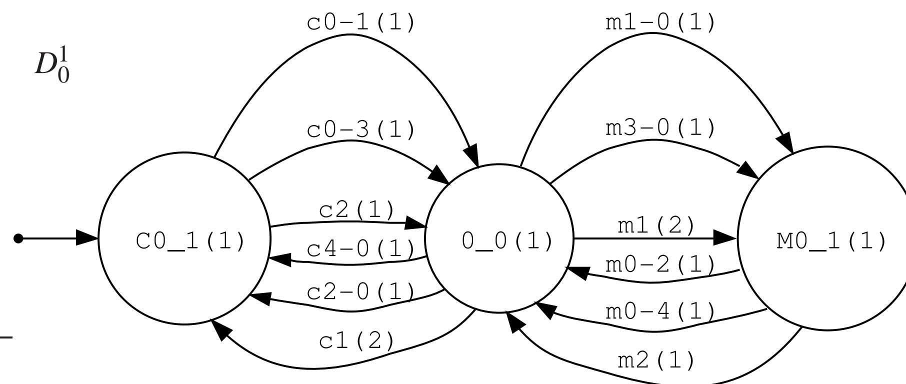


Figure 4.4: Specification automaton $D_0^1$ for room 0 on floor 1.

The coordination between the single room models on one hand, and between the models and the specification on the other hand is always realized by regarding the room changing events.

### 4.1.3 Computing the supervisor

Before being able to compute the supervisor, the room models have to be composed by parallel composition. Eventually, a plant generator and a specification generator shall be available. As this plant generator for level 1 consists of 36 states (as well as the plant generator for level 2), the composition

$$C^1 = C^1_0 \parallel C^1_1 \parallel C^1_2 \parallel C^1_3 \parallel C^1_4 \parallel \hat{C}^1$$

for the cat on level 1 shall be presented in automaton $C^1$ (Figure 4.5) as an intermediate result. The cat's possible paths can easily be retraced. When the cat goes up to level 2, we arrive at state 4 and the automaton is not able to relate to the cat's movement in level 2. We return to the initial state 1 when the cat descends.

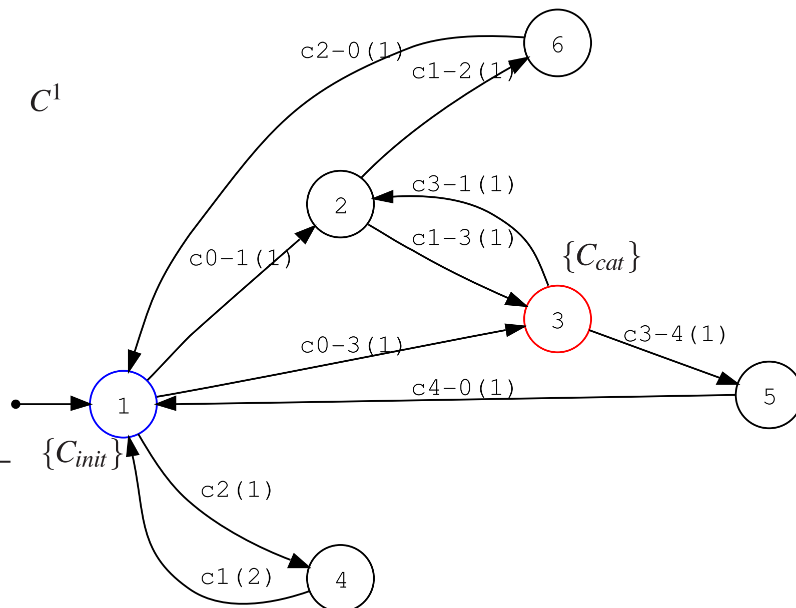


Figure 4.5: Composition $C^1$ for all room models concerning the cat and level 1.

The composition for the other level and for the mouse's both level compositions is realized in the same way.

The specification composition for level 1 results as

$$D^1 = D^1_0 \parallel D^1_1 \parallel D^1_2 \parallel D^1_3 \parallel D^1_4.$$

With those compositions available, we are able to compute two strongly nonblocking supervisors, one for each level. They both consist of 27 states, what implements that the

supervisor deleted 9 states on every floor. They either were forbidden by the specification or resulted in blocking. The supervisor for level 1 is shown in Figure 4.6.

The parallel composition $D^1 \parallel D^2$ of both modular supervisors results in a nonblocking generator with 82 states. Hence, the two modular supervisors are nonconflicting and can separately be implemented to achieve strongly nonblocking control while obeying the given specification. Furthermore, it turns out that a monolithic supervisor for the control problem under study also has 82 states. This means that the modular supervisors achieve maximal permissive control which is due to the fact that all shared events are controllable [LW02].

Altogether, the insertion of colored markings and the use of a modular structure lead to two crucial advantages. First, the completion of several tasks which is indicated by appropriate color labels can be guaranteed by the designed supervisors, and second, the usage of multiple modular supervisors reduces the state size of the implemented supervisors. For this example, the number of states in $C^1$ and $C^2$ amount to 54, whereas the monolithic supervisor has 82 states. For more complex systems, this proportion between the respective state numbers can be much higher.

## 4.2 Fischertechnik Production Plant

The Fischertechnik production plant model at the Chair of Automatic Control shown in Figure 4.7 is predestinated to demonstrate the use of multitasking hierarchical control. It consists of several conveyor belts transporting parts from a stack feeder to two machine heads with a drill on each and finally brings the finished workpieces to a deposition area. All machine parts can thereby be modeled for themselves and several levels of specification can be used. Furthermore, the completion of multiple tasks is ensured by colored marking.

### 4.2.1 General Description of the Production Plant and the Chosen Part for this Example

The description of the plant operation follows the notation in Figure 4.8. Workpieces enter the plant from a stack feeder sf. A sensor detects if there are any workpieces and if so, they are transported by conveyor belt cb1 and other ones to both machine heads mh1 and

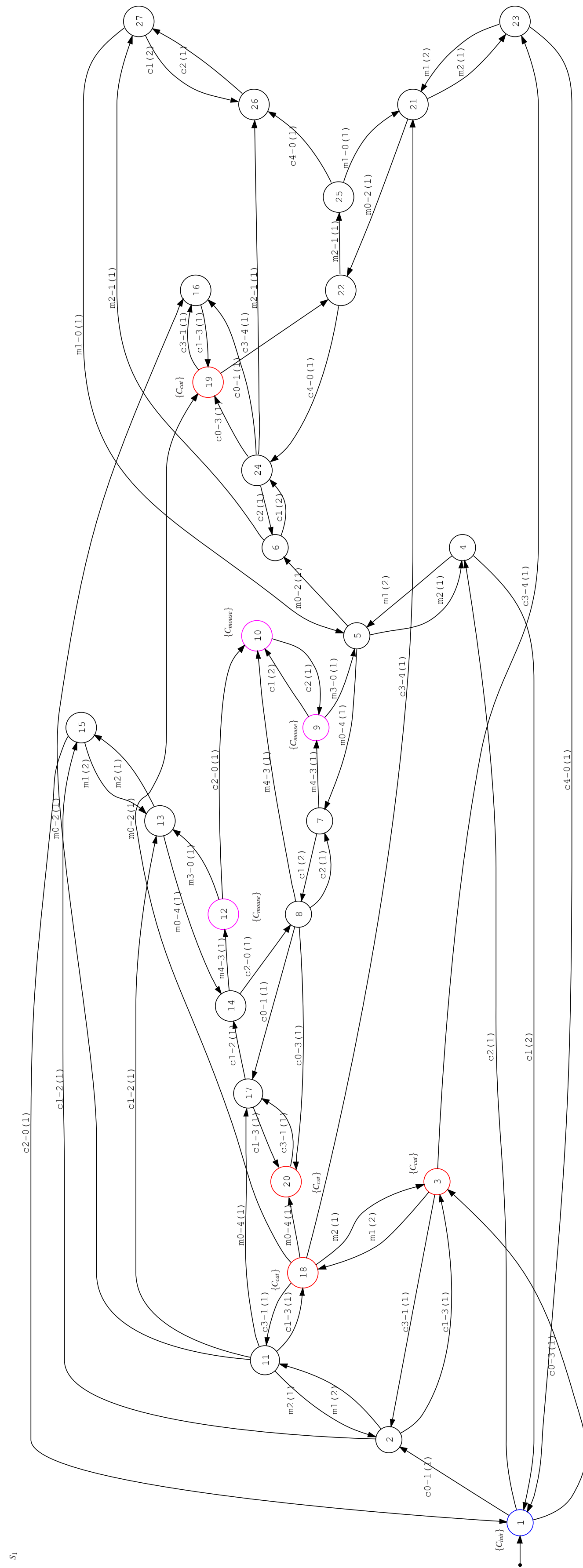


Figure 4.6: Supervisor $S_1$ for floor 1 of the cat and mouse in a maze example.

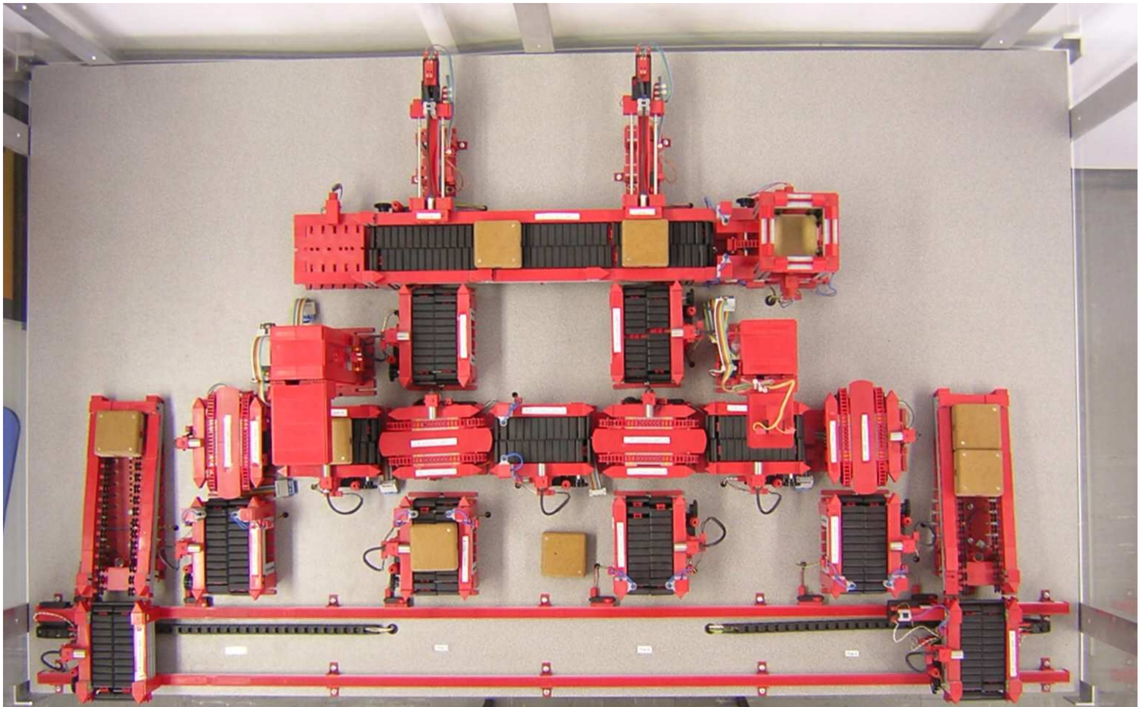

Figure 4.7: Picture of the Fischertechnik production plant.

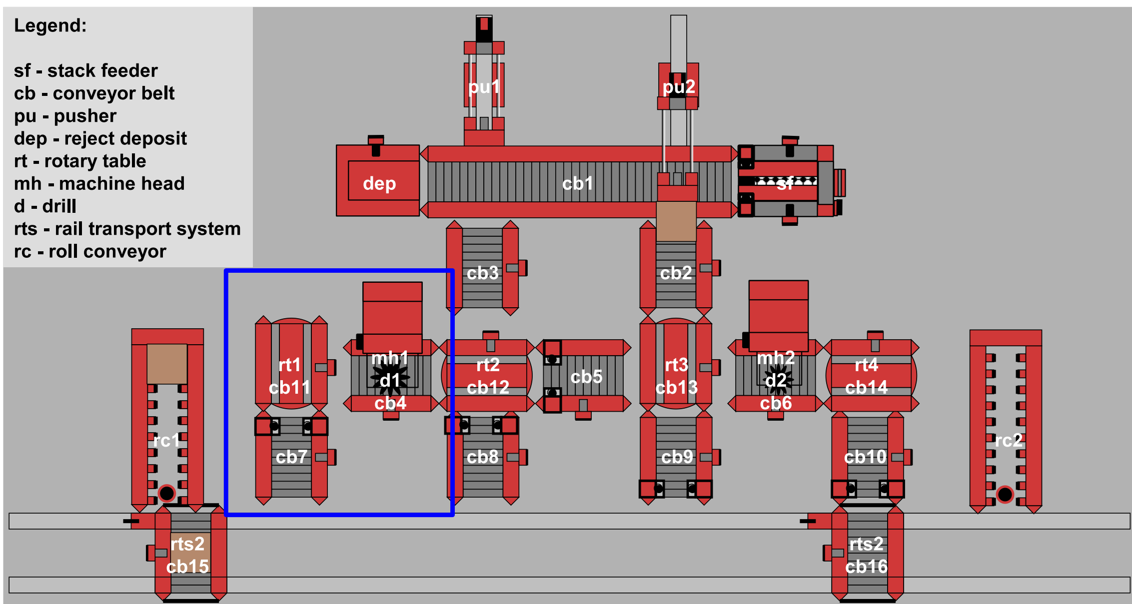


Figure 4.8: Fischertechnik Production Line: Schematic Overview.

mh2 where processing takes place. At the end, the workpieces are stored in one of two deposition areas (rc1 and rc2).

All elements of the plant are labeled consistently. The stack feeder is specified as *sf*, the conveyor belts as *cbX*. The machine heads with the drills are abbreviated as *mhXdY*, the rotary tables as *rt* and the workpiece detection elements which recognize the kind of workpiece as *wpdetX*. X or Y, respectively, stand for the particular number of an element and are necessary for an unambiguous labeling. The shared events between the separate conveyor belts which coordinate the behavior of the composed plant are *cbX-cbY* where X describes the origin and Y the destination of the respective workpiece.

In this work, we regard the framed part of the plant scheme in Figure 4.8. It consist of conveyor belt cb4 which is combined with a machine head (mh1d1), conveyor belt cb11 which also takes the role of a rotary table (rt1) and conveyor belt 7 (cb7), which brings along a workpiece detection element. It allows to distinguish workpieces of separate types when they are going from cb7 to cb11 (`cb7-cb11`) or from cb11 to cb7 (`cb11-cb7`). Thus, workpiece characteristics can be regarded and different behaviors for different types of workpieces can be considered in the model and the specification of the conveyor belt where they arrive.

In our study, we consider two types of workpieces: Workpiece 1 denotes a workpiece which is completely and correctly processed, whereas workpiece 2 stands for workpieces which have not yet been processed or which are of insufficient quality. Therefore, those workpieces have to be transported to cb4 where processing or reprocessing, respectively, takes place.

Workpieces which are destined for the chosen part of the plant may either arrive at conveyor belt cb12 or cb15. Those from cb12 shall be drilled and transported via cb11 and cb7 to cb15 (Figure 4.9) or be directly returned to cb12 without being processed. If a workpiece is of bad quality, which is detected when being moved from cb11 to cb7, it shall return to cb4, be reprocessed, and afterwards be delivered to cb12. Workpieces arriving at cb7 and coming from cb15 also shall go to cb4, be processed there and finally leave the section towards cb12. If one of those workpieces is detected as already finished when leaving cb7 towards cb11, which means it is already processed and of good quality, it need not be drilled any more and therefore shall be stopped at cb11 and be returned to cb15.

Colored marking assures that all tasks can be terminated. In this case, it is determined that all workpieces can be drilled and that all conveyor belts can return to an empty state.

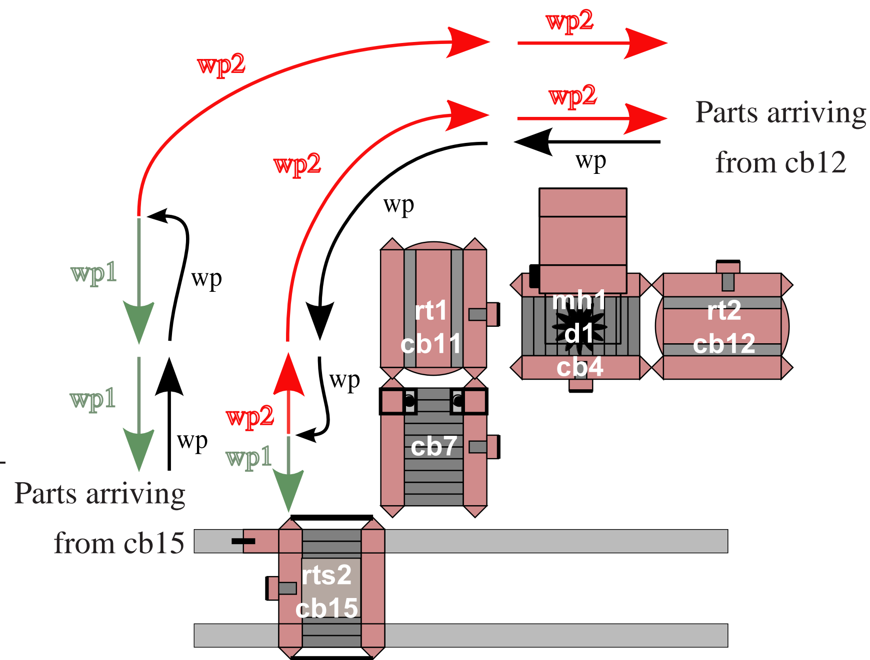


Figure 4.9: Paths of workpiece transportation in cb4 - cb7 - cb11.

Of course, the supervisor to develop has to guarantee strongly nonblocking and shall ensure that each workpiece can be drilled.

## 4.2.2 Modeling and Specification

### Level 0

The modeling of the considered production plant's part is carried out in a hierarchical way. The hierarchy level is attached to every automaton's name in square brackets. On the first level all single elements of this section are described as automata. Doing so, we get automata cb4[0], cb7[0], and cb11[0] which denote the conveyor belts, mh1d1[0] which stands for the machine head and the drill, and rt1[0] which denotes the rotary table. wpdet7[0] or wpdet11[0] are special parts that detect the status of a workpiece arriving at cb7 or cb11, respectively. Note, that detection of a workpiece is only possible when it is transported from cb7 to cb11 or the other way round.

The colors introduced for the observed part of the plant are set that way, that every conveyor belt is able to return to an empty state. Furthermore, one color which denotes that a workpiece has just been drilled is introduced. Thus, the overall color set results in $C = \{C_{drilled}, C_{cb4}, C_{cb7}, C_{cb11}\}$.

In addition to the mentioned models, there is cb7cb11[0] (Figure 4.10), which makes sure that in cb7 or cb11 no illegitimate states are possible. For instance, let there be a workpiece transported from cb11 to cb7 which is detected as one of insufficient quality. It would be transported back to cb11. Consequently, when returning to cb11, we already know that it has not yet been processed accordingly and therefore, we do not need to regard the possible behavior for a workpiece 1. The model cb7cb11[0] takes care of all those possible dependencies and reduces the state space of the resulting model accordingly. Nevertheless, it does not constrain the system in an undesirable way. It does not say what events to allow after a special type of workpiece appeared. It simply excludes impossible events that unnecessarily enlarge our model.

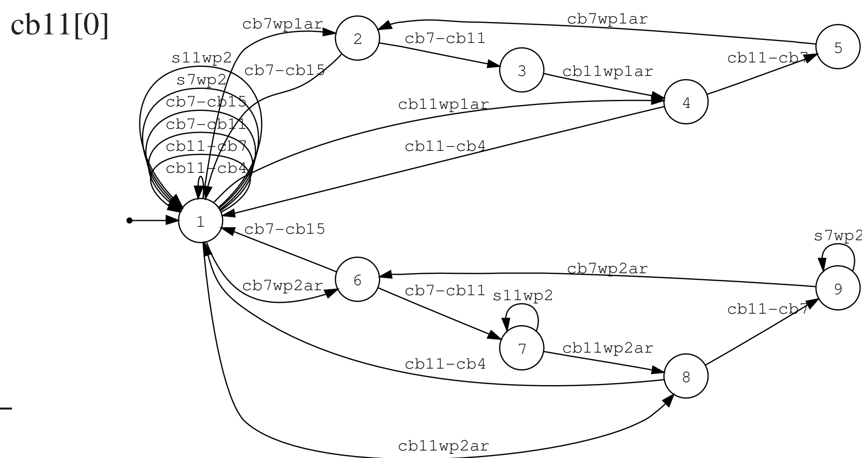


Figure 4.10: Model of the automaton linking cb7 and cb11.

As an example, of our proceedings, the hierarchical design of cb11 and the rotary table rt1 shall be regarded (Figure 4.11). First, the model for the conveyor belt cb11[0] is realized. The shared events `cb7-cb11` and `cb4-cb11` lead to a movement of the conveyor belt in the required direction (`cb11+x+y` or `cb11-x-y`). The event `t_cb11` symbolizes the time passing until the workpiece arrives (`cb11wpar`, `cb11wp1ar`, or `cb11wp2ar`). It is necessary, as the movement could be stopped before the occurrence of an arriving event, which leads back to the initial state, or afterwards. In the latter case, the procedure for bringing the workpiece away has to follow. It is similar to the described event chain, only the arriving events are replaced by the leaving ones `cb11wplv4` and `cb11wplv7`.

The according specification consists of two parts and is shown below that automaton. cb11[0]_spec1 defines the correct functionality of cb11 considering in which direction workpieces are transported. cb11_spec2[0], however, controls the handling of different types of workpieces. Both specifications are combined by parallel composition to a sin-

cb11[0]

cb11[0]_spec1

$\{C_{cb11}\}$

cb11[0]_spec2

cb11[1]

$\{C_{cb11}\}$

cb11[0]_sup

$\{C_{cb11}\}$

Figure 4.11: Model, specifications and supervisor for cb11 on level 0.

gle one. Together with the model cb11[0] the supervisor cb11[0]_sup in Figure 4.11 is computed.

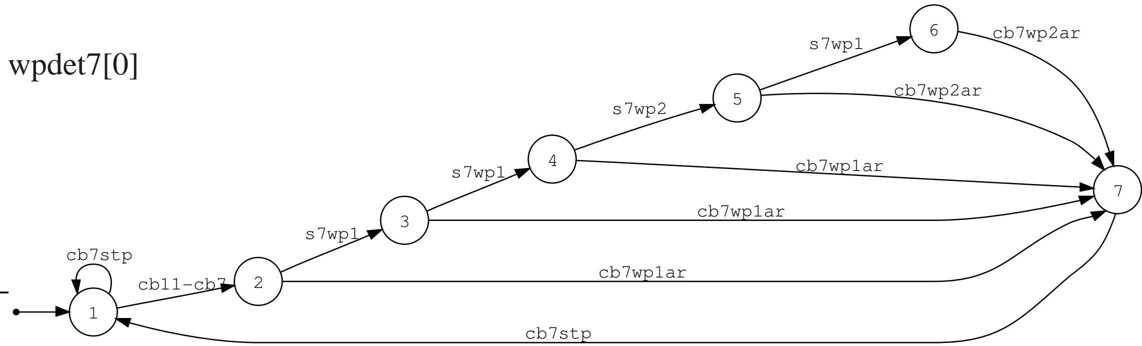


Figure 4.12: Workpiece detection element modeled for cb7.

In our plant model, the workpieces are distinguished depending on the number of magnetic pins located on the workpiece (Figure 4.12). They either have 0, 1, or 2 pins which means they represent a workpiece of the first type, or 3 or 4 pins what makes them a workpiece 2. After the event `cb11-cb7` which indicates that a workpiece will arrive, the number of sensor signals `s7wp1` and `s7wp2` that occur until the conveyor belt stops are counted. `s7wp1` and `s7wp2` denote the same physical event, but the different labeling is chosen to support the later abstraction step.

Before being able to abstract unramified process chains in cb11[0]_sup, the validity of the observer condition for this supervisor and the abstraction alphabet $\Sigma_{proj} = \{$`cb4-cb11`, `cb11-cb4`, `cb7-cb11`, `cb11-cb7`, `cb11wpar`, `cb11wp1ar`, `cb11wp2ar`, `cb11stp`$\}$ has been verified with a positive result. Then, the following projection step for cb11[0]_sup with $\Sigma_{proj}$ leads to the abstracted automaton cb11[1] shown in Figure 4.11. The other plant elements such as the machine head with the drill mh1d1 or the conveyor belts cb4 and cb7 are dealt with the same manner (see Appendix A.1).

### Level 1

On level 1, the combination of the rotary table and the conveyor belt is realized. The parallel composition of cb11[1] and rt1[1] (Figure 4.13), which represents the abstraction of the controlled rotary table, results in the level 1 model cb11rt1[1]. `rt1mvx` and `rt1mvy` describe the rotation in x or y direction, `rt1stp` occurs if the rotary table stops.

The corresponding specification also consists of several components. cb11[1]_spec thereby assures that the conveyor belt does not move when the table rotates. rt1[1]_spec1

takes care that the conveyor belt only can start when the table is not rotating. The second specification for the rotary table, rt1[1]_spec2, describes the necessity of a rotation depending on the shared events. Building the parallel composition of those three specifications allows us to find a supervisor cb11[1]_sup for the second abstraction level.

After having checked that the observer condition is fulfilled for that supervisor and the corresponding abstraction alphabet $\Sigma_{proj} = \{$`cb4-cb11`, `cb11-cb4`, `cb7-cb11`, `cb11-cb7`, `cb11wpar`, `cb11wp1ar`, `cb11wp2ar`, `s11wp2`$\}$, the abstraction step is carried out for cb11[1]_sup and only the shared events that make up $\Sigma_{proj}$ are left in the abstracted generator cb11[2]. The rotary table events do not longer appear on this level. The abstracted generator version cb4[2] for conveyor belt cb4 is computed in the same way (see Appendix A.1).

**Level 2 and Final Result on Level 3**

All level 2 abstractions of the single blocks around cb4, cb7 and cb11 can in a further step be composed to the model cb4cb7cb11. To ensure nonblocking, a specification with a single state and self-loops for all possible events is created. For this one, the supervisor will only take care of nonblocking, as the specification does not constrain the plant's behavior.

To attain the controlled level 3 plant model another abstraction step is carried out. The resulting generator then consists of 37 states.

### 4.2.3 Hierarchical Structure

The hierarchical structure which results from that procedure is shown in the subsequent diagrams (Figures 4.14 and 4.15).

As already described, the specification cb11[0]_spec for cb11 consists of two parts which are composed by parallel composition. The resulting supervisor for it and the corresponding model cb11[0] is then abstracted to cb11[1]. On this level, the abstractions for the conveyor belt, the rotary mechanism and for the workpiece detection elements are composed. With the appropriate specification for the first level (Figure 4.14, right side) the supervisor for level 1 is computed and afterwards abstracted to the next higher level.

In the same manner, the other conveyor belts cb7 and cb4 including their respective machine parts are modeled and specified. cb7's workpiece detection element is, as well as

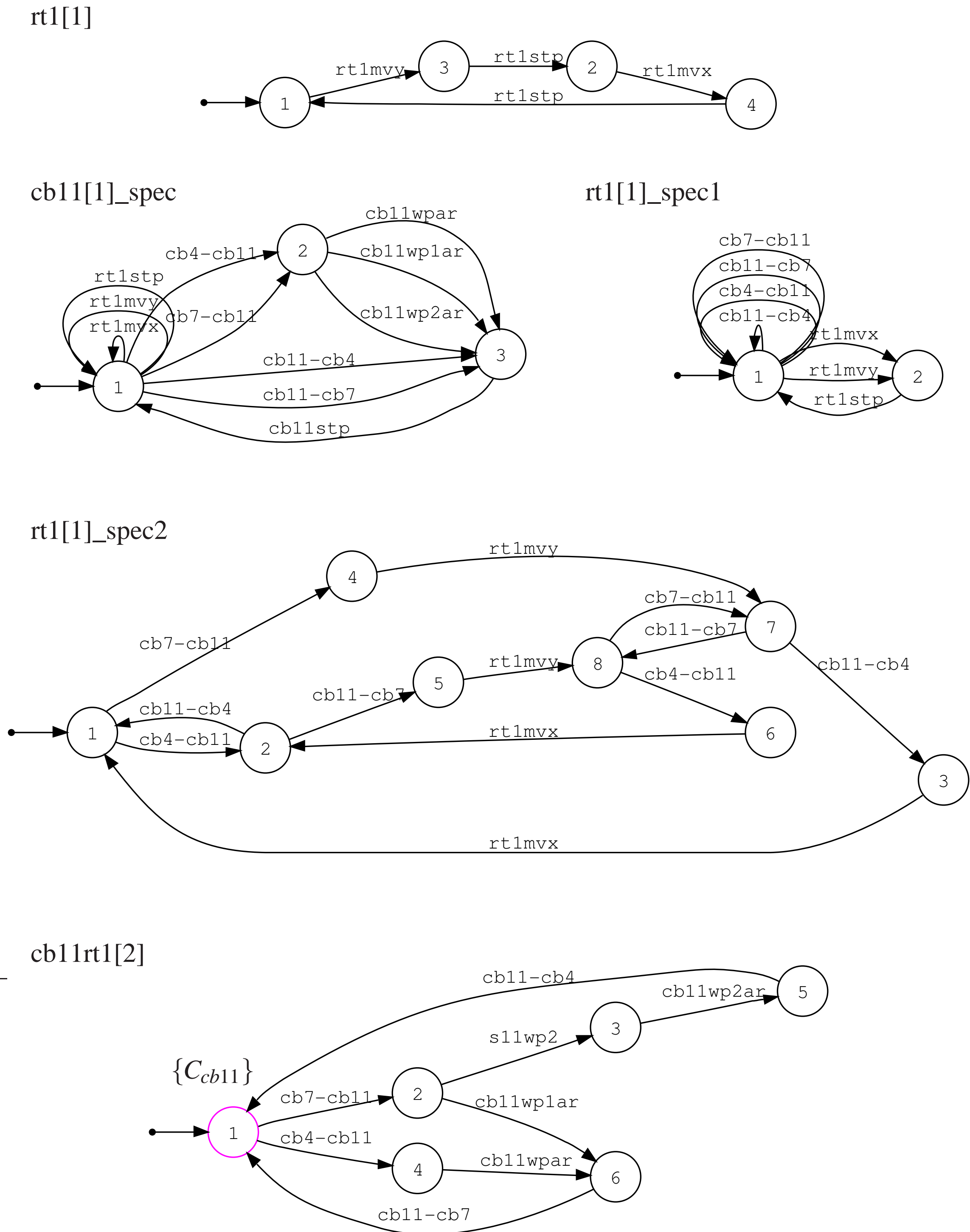


Figure 4.13: Model, specifications and supervisor for cb11 and rt1 on level 1.

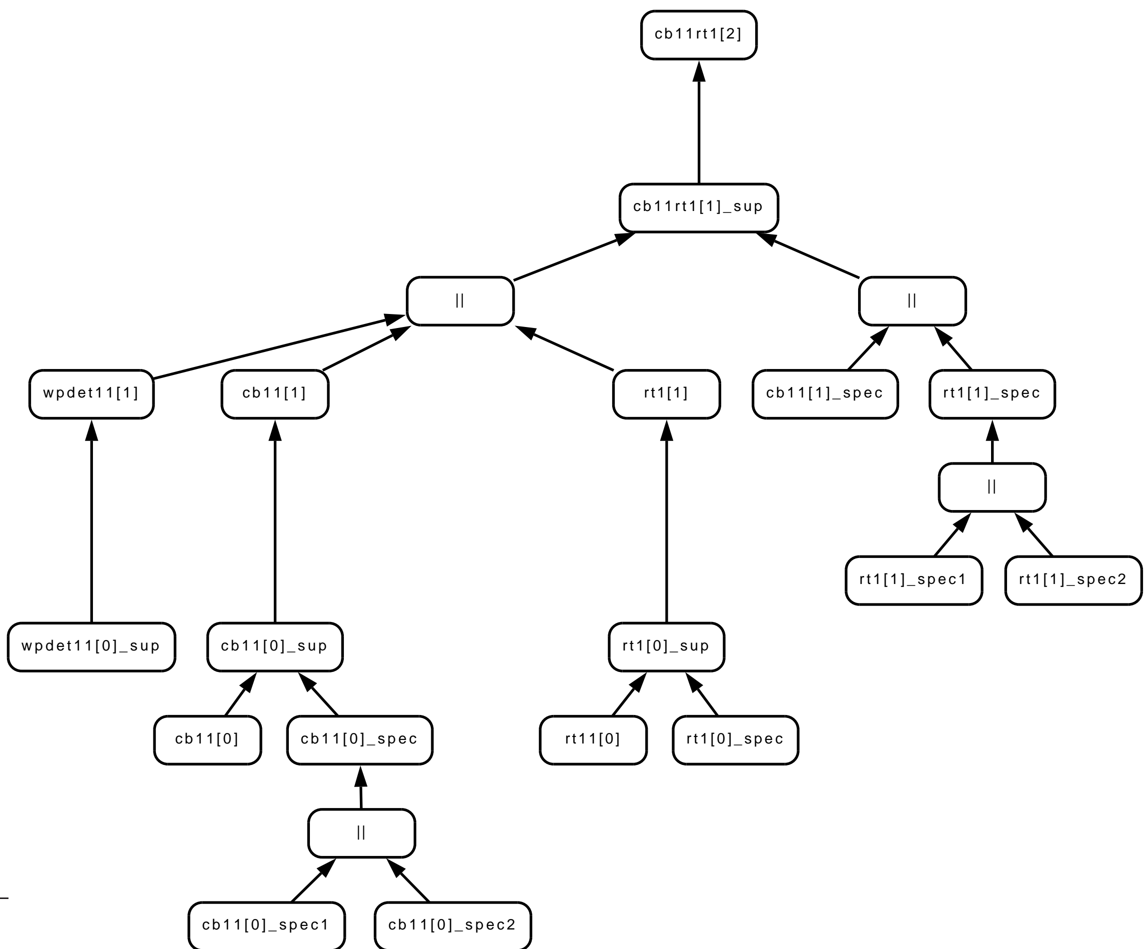


Figure 4.14: Hierarchical structure for level 1 and 2 for cb11.

with cb11, composed with the level 1 supervisor. As there are not further parts such as drills or a rotary table to regard, no level 1 specification is necessary. Hence, the composition's result cb7wpdet7[1], the level 2 abstractions cb4cb7cb11[2] and cb11rt1[2], and the model cb7cb11[0] are composed to the plant part's model. The appropriate specification only consists of one state and self-loops for all events. Consequently, the supervisor to synthesize shall only guarantee strongly nonblocking and, moreover, does not constrain the model any further. The final result cb4cb7cb11[3] arises from a further abstraction step and can be used together with the rest of the overall plant.

The complete hierarchy tree can be found in Appendix A.2, the supervisor cb4mh1d1[2] and the separate models and specifications used in Appendix A.1. The automaton for the top-level abstraction cb4cb7cb11[3] with its 37 states is provided in Appendix A.2-17.

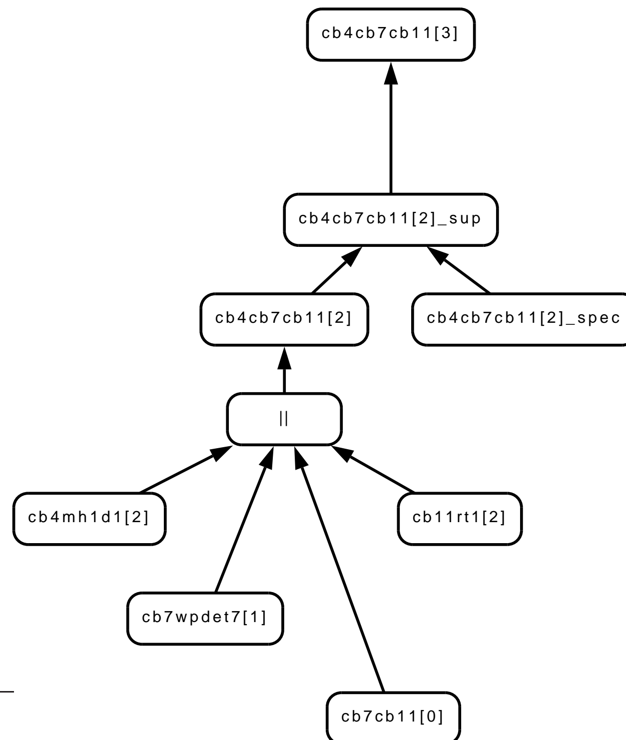


Figure 4.15: Combination of all components to the strongly nonblocking supervisor cb4cb7cb11[2]_sup which is abstracted to cb4cb7cb11[3] in the last step.

The Fischertechnik production plant example clarifies the necessity of several markings to ensure the completion of multiple tasks, what in this case meant the production of all workpieces (color $C_{drilled}$) and that all of them can be carried away again (colors $C_{cb4}$, $C_{cb7}$, and $C_{cb11}$). Furthermore, this example verifies that the state space reduction can be enormous for complex systems. A monolithic supervisor for the chosen part of the plant would possess 27,614 states, whereas the combination of low- and high-level

supervisors only leads to a sum of 277 states. Moreover, the monolithic synthesis requires the evolution of a plant automaton with 2,336,400 states, while the largest automaton in the hierarchical multitasking control has 254 states.

Together, this approach reduces the computational costs and the memory consumption in the control unit very effectively. At this point, it has to be mentioned that the reduced complexity of the supervisor computation and implementation is accompanied with a possible loss of maximal permissiveness. In our example case, the parallel composition of all hierarchical supervisors leads to an automaton with 25,904 states which implies that the control is more restrictive than the monolithic supervisor.

# Chapter 5

# Conclusion

This thesis first summarized the theoretical background of the multitasking supervisory control theory. Therefore, colored marking generators (CMG) were introduced as a model that represents multiple tasks by states with different colors. Based on such CMGs, relevant properties such as strongly nonblocking have been introduced to express nonblocking behavior w.r.t. different generator colors. Additionally, several operations for CMGs including the colored marking parallel composition, the supervisor synthesis for CMGs and tools for the hierarchical supervisory control of colored marking generators have been presented.

As the first contribution of this thesis, the realization of necessary data structures and the appropriate methods in a plugin for the libFAUDES software library was described. In particular, the class *TmtcGenerator* was implemented as a representation of a CMG. It extends the *cGenerator* class which is suitable for supervisor synthesis in the Ramadge/ Wonham framework by state attributes that serve for saving state colors. The methods of this class allow the user to insert colors into a multitasking generator, to delete them, and to analyze and modify the respective generator. Furthermore, methods for generator input and output are included. Additionally, the plugin provides functions for the parallel composition of several CMGs, for state minimization, and for the supervisor synthesis. Despite its complexity, the plugin offers simple user interfaces and thus allows its users to easily synthesize and analyze CMGs without knowing implementation details.

Finally, the multitasking plugin was used to implement two examples, the cat and mouse in a maze and the Fischertechnik production plant. Both examples show the applicability of CMGs, as they represent systems with several parallel tasks. In both cases, strongly nonblocking behavior can comfortably be specified with colored markings and the re-

spective supervisors can be synthesized algorithmically.

Simultaneously using hierarchical design methods even enables the modeling of systems with very high complexity, as separate system components can be modeled and controlled on their own. Accordingly, the corresponding specifications can be divided into several generators. This approach effectively increases the project's clarity and thus helps to avoid design faults. Furthermore, the number of states is dramatically reduced what minimizes computational costs and the memory usage in the respective control units. For instance, the analyzed part of the Fischertechnik production plant results in 27,614 supervisor states when choosing a monolithic approach. In contrast, the appropriate hierarchical system uses 10 individual supervisors with a sum of 277 states.

# Bibliography


[CL99] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.

[dox08] Doxygen - Source Code Documentation Generator Tool, 2008. Homepage: http://www.doxygen.org.

[dQC04] M.H. de Queiroz and J.E.R. Cury. Multi-tasking supervisory control of discrete event systems. *WODES*, 2004.

[dQC05] M. H. de Queiroz and J. E. R. Cury. Modular multitasking supervisory control of composite des. In *In Proc. of the 16th IFAC World Congress, Prague, Czech Republic*, 2005.

[gra08] Graphviz - Graph Visualization Software, 2008. Homepage: http://www.graphviz.org.

[lgp07] GNU Lesser General Public License, 2007. Homepage: http://www.gnu.org/licenses/lgpl.html.

[lib08] The libFAUDES Software Library, 2008. Homepage: http://www.rt.eei.uni-erlangen.de/FGdes/faudes/index.php.

[LW02] S-H. Lee and K.C. Wong. Structural decentralised control of concurrent des. *European Journal of Control*, 35:1125–1134, October 2002.

[RW87] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event systems. *SIAM J. Control and Optimization*, 25:206–230, 1987.

[RW89] P.J. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77:81–98, 1989.

[Sch05] K. Schmidt. Hierarchical control of decentralized discrete event systems: Theory and application. *PhD-thesis, Lehrstuhl für Regelungstechnik, Universität Erlangen-Nürnberg*, 2005. Download: http://www.rt.eei.uni-erlangen.de/FGdes/publications.html.

[SQC07] K. Schmidt, M. H. Queiroz, and J. E. R. Cury. Hierarchical and decentralized multitasking control of discrete event systems. *IEEE Conference on Decision and Control*, 2007.

# Appendix A

# Fischertechnik Production Plant

## A.1 Models and Specifications for Conveyor Belt 4 and its additional components

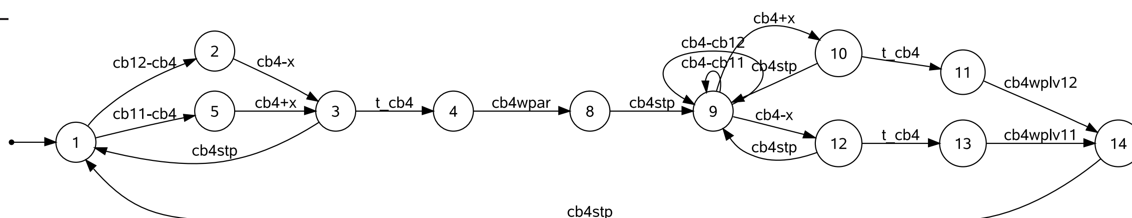


Figure A.1-1: Conveyor belt 4: cb4[0]

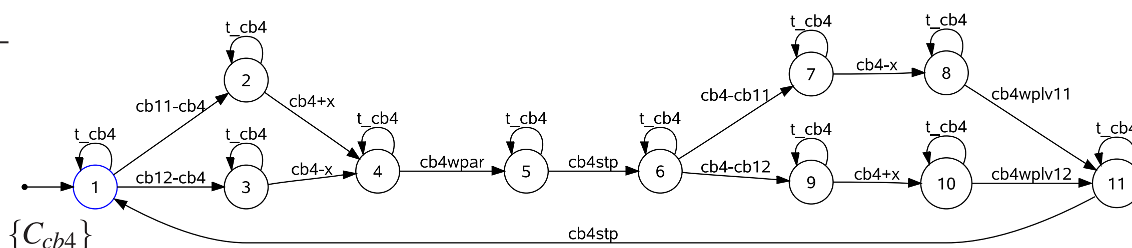


Figure A.1-2: Conveyor belt 4: cb4[0] _spec1

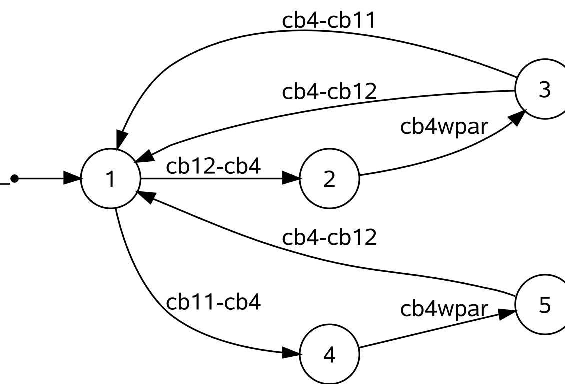


Figure A.1-3: Conveyor belt 4: cb4[0]_spec2

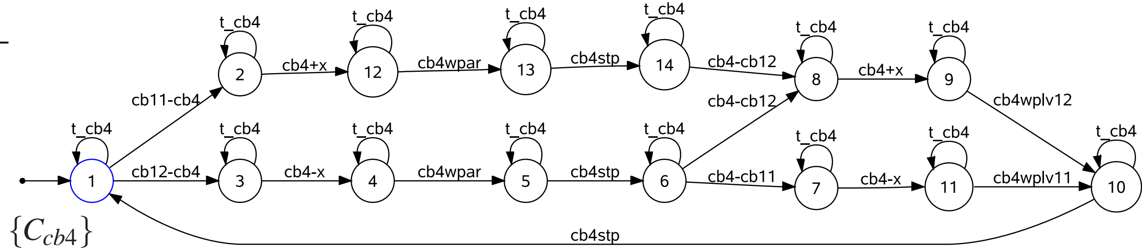


Figure A.1-4: Conveyor belt 4: cb4[0]_spec

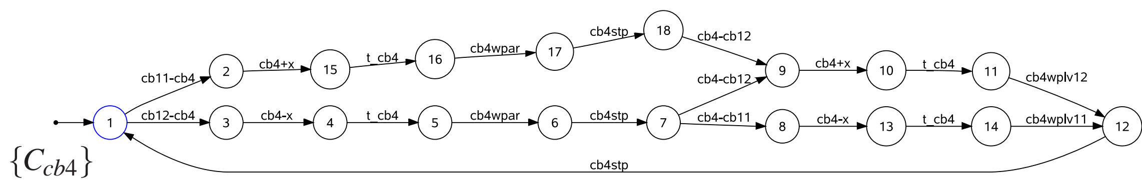


Figure A.1-5: Conveyor belt 4: cb4[0]_sup

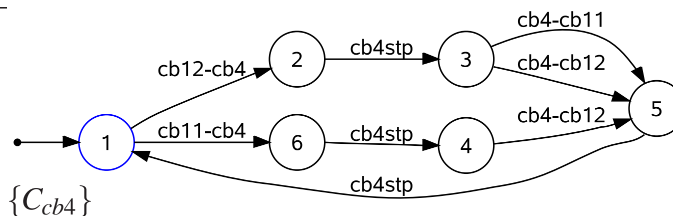


Figure A.1-6: Conveyor belt 4: cb4[1]

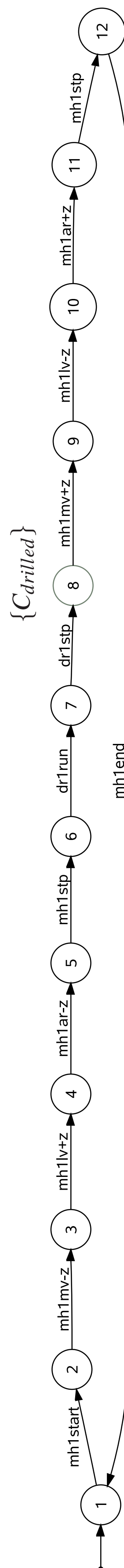


Figure A.1-7: Machine head and drill: mh1d1[0]_sup

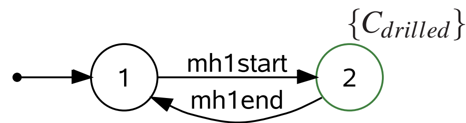


Figure A.1-8: Machine head and drill: mh1d1[1]

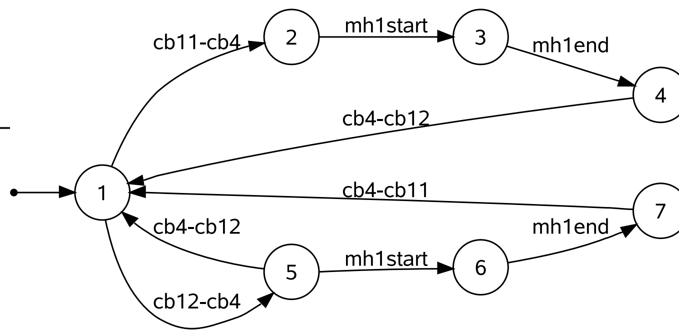


Figure A.1-9: Machine head and drill: mh1d1[1]_spec

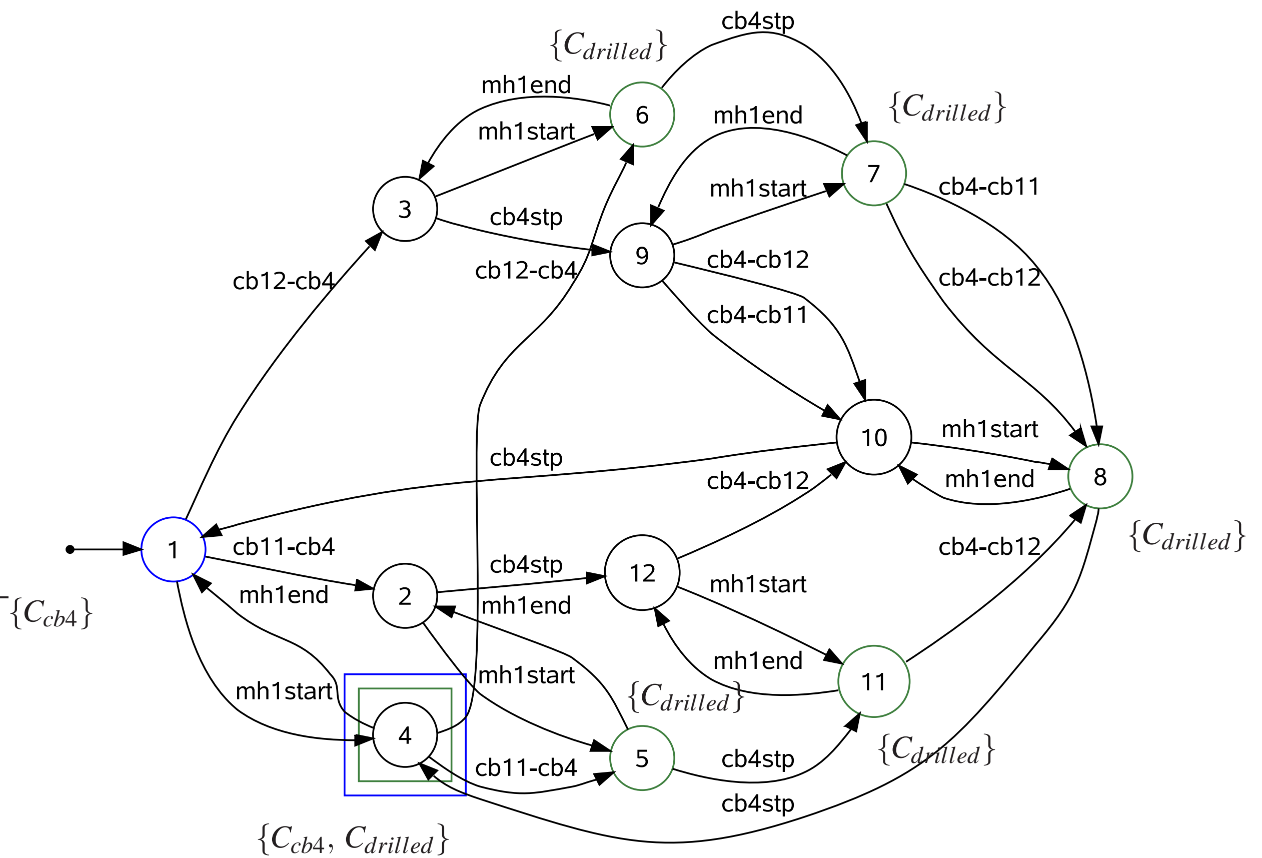


Figure A.1-10: Conveyor belt 4 + machine head with drill: cb4mh1d1[1]

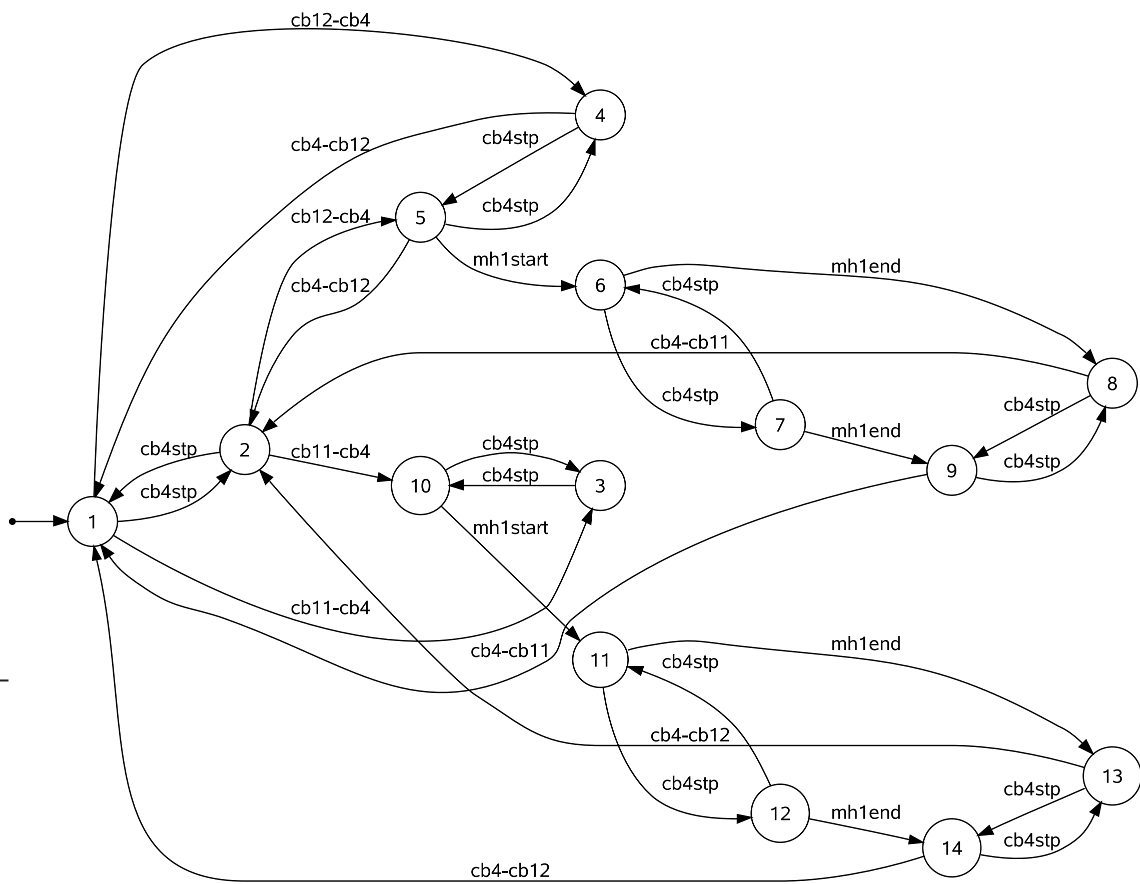


Figure A.1-11: Conveyor belt 4 + machine head with drill: cb4mh1d1[1]_spec

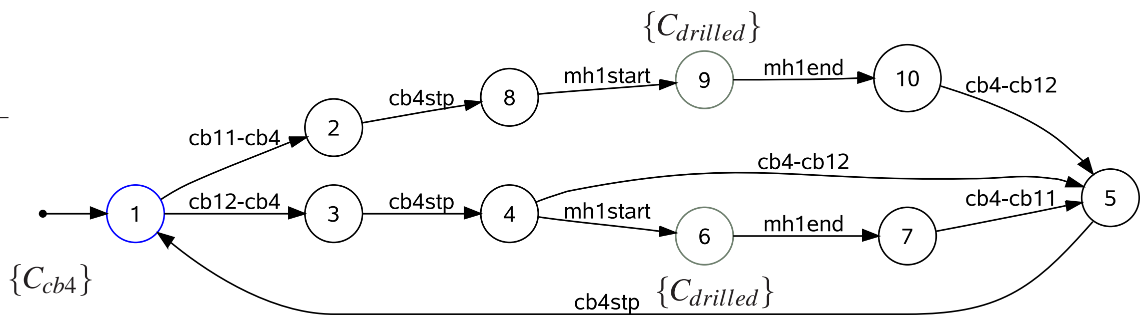


Figure A.1-12: Conveyor belt 4 + machine head with drill: cb4mh1d1[1]_sup

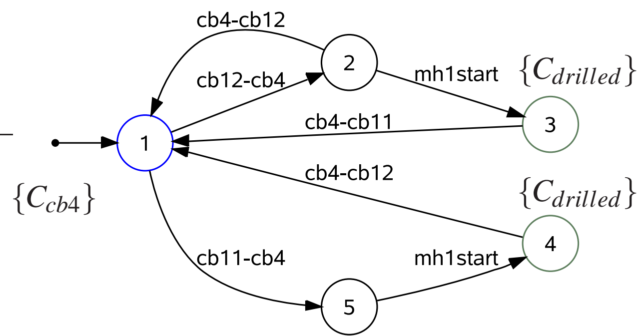


Figure A.1-13: Conveyor belt 4 + machine head with drill: cb4mh1d1[2]

## A.2 Hierarchy Diagrams

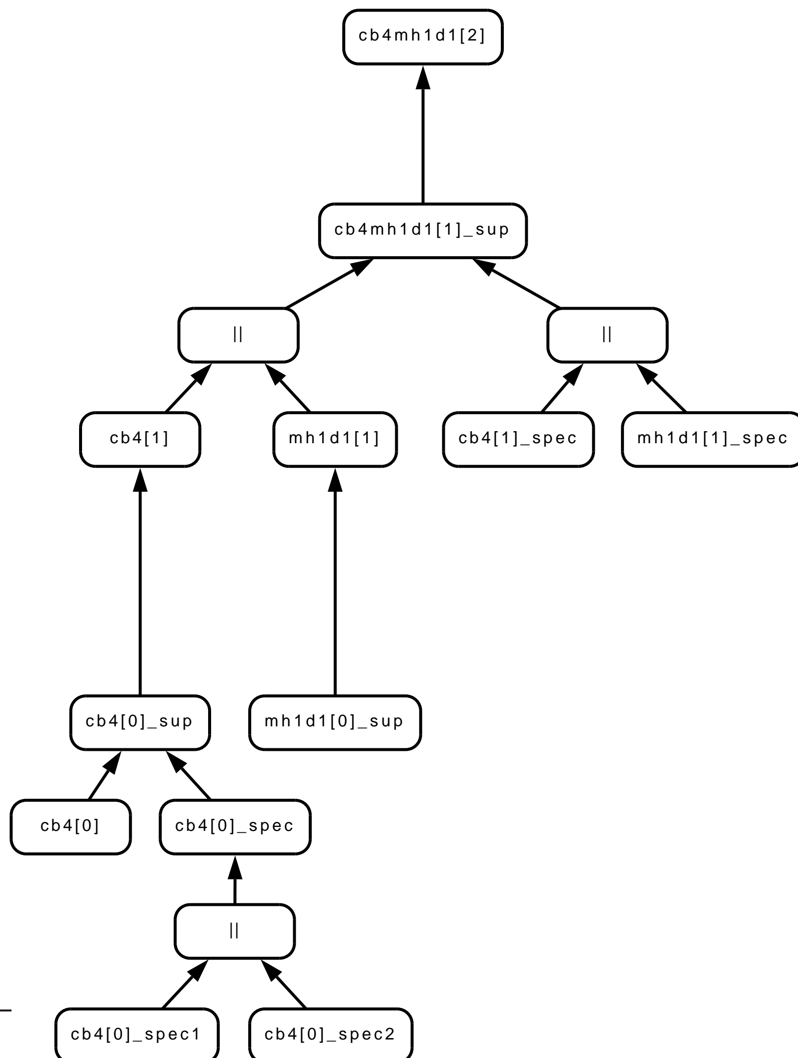


Figure A.2-14: Conveyor belt cb4 abstracted to level 2

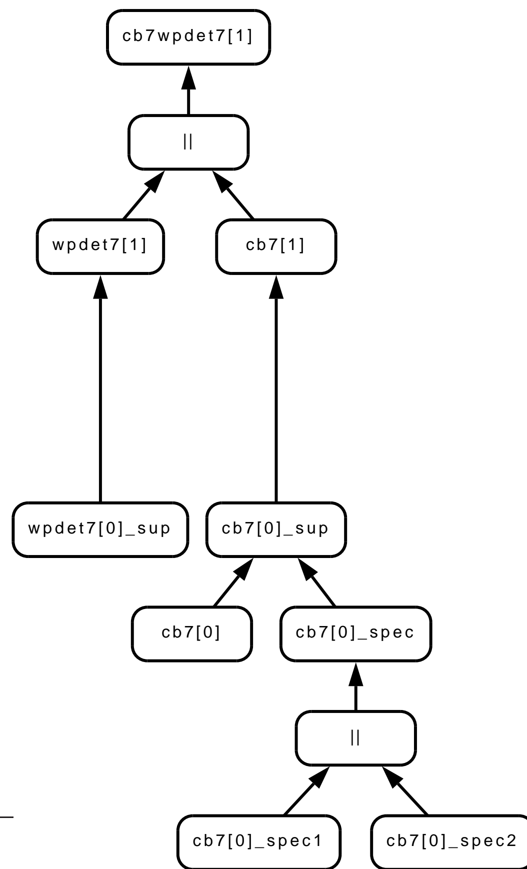


Figure A.2-15: cb7 and the workpiece detection element combined on level 1

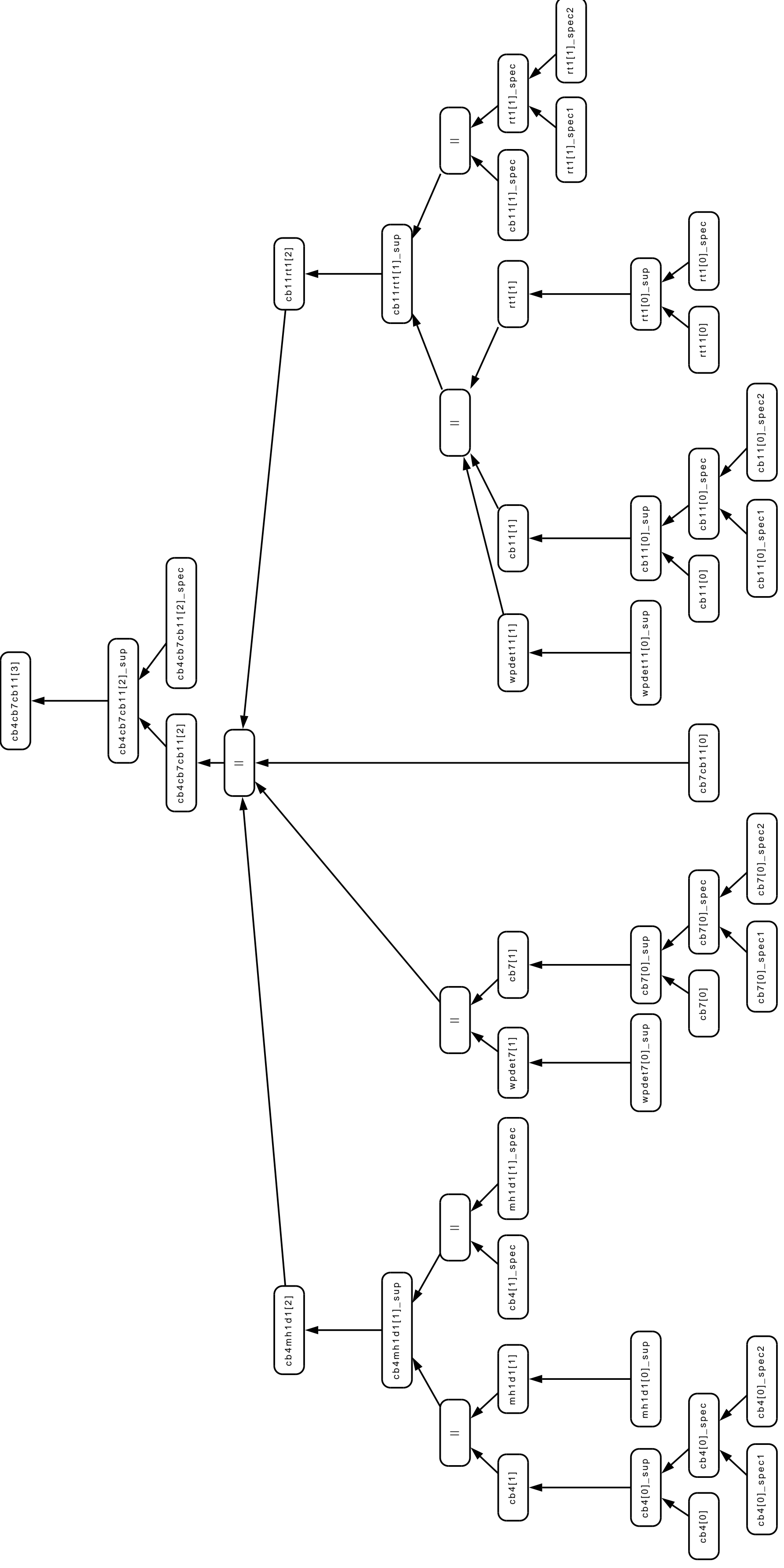


Figure A.2-16: Complete hierarchy diagram for the considered part of the Fischertechnik production plant model

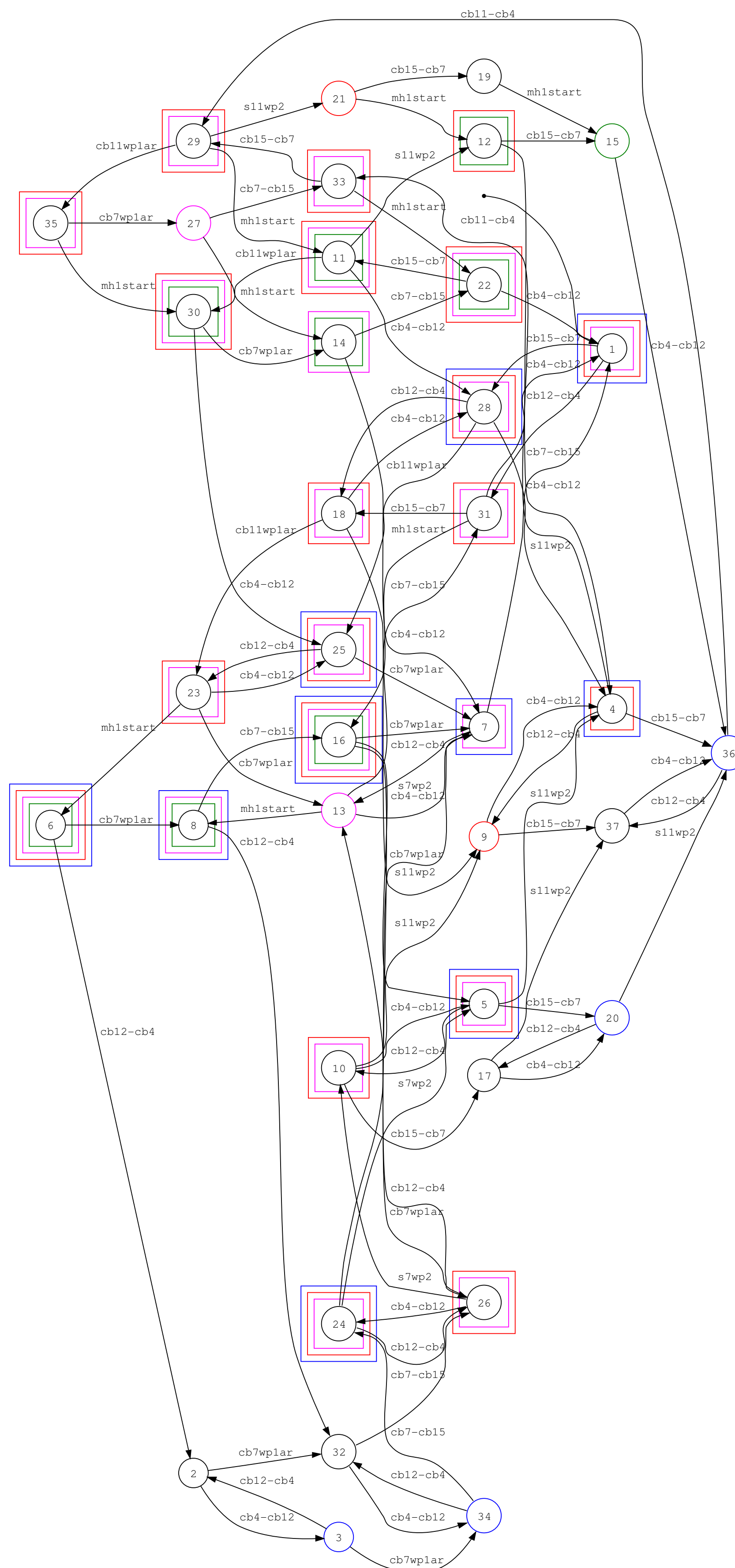


Figure A.2-17: Resulting level 3 abstraction with 37 states. Color meanings: blue = cb4 empty, red = cb7 empty, pink = cb11 empty, green = workpiece drilled